DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Estudio comparativo de performance de implementaciones de una simulación de flujo de fluidos en C y en ASM x86-64 con procesamiento vectorial

5 de octubre de 2017 — Organización del Computador II

Grupo: Ariane 5

Integrante	LU	Correo electrónico
Greco, Luis	150/15	luifergreco@gmail.com
Hertzulis, Nicolás	811/15	nicohertzulis@gmail.com
Ramos, Ricardo	841/11	riki_german@yahoo.com.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción **2**

2. Desarrollo **2**
2.1. Función solver_lin_solve 2
2.2. Función solver_set_bnd 5
2.3. Función solver_project 5
2.4. Eficiencia C vs paralelismo simd ASM 6

3. Resultados **7**
3.1. Función solver set bnd 7
3.2. Función solver lin solve 8
3.3. Función solver project 8

4. Conclusión **9**
4.1. Implementación en código ASM 9
4.2. Eficiencia C vs paralelismo simd ASM 9

1. Introducción

En computación, un procesador vectorial es una unidad central de proceso que implementa un conjunto de instrucciones que operan sobre arreglos de una dimensión de tamaño fijo, llamados vectores, a diferencia de procesadores escalares, cuyas instrucciones operan únicamente sobre datos individuales.

Algunos programas informáticos pueden ser implementados en lenguaje ensamblador utilizando exclusivamente instrucciones escalares o utilizando también instrucciones vectoriales. Para estos programas el procesamiento vectorial constituye una optimización dado que se percibe un incremento en la performance (i.e. tiempo neto de procesamiento).

Por otro lado, los compiladores modernos de lenguajes de alto nivel, como GNU C Compiler (GCC) incluyen la posibilidad de realización de diversas optimizaciones que se aprecian en el código ensamblado, entre las que se encuentra la utilización de instrucciones de procesamiento vectorial cuando es posible.

Para el estudio comparativo utilizamos una simulación de flujo de fluidos basada en las ecuaciones de Navier-Stokes escrita en lenguaje C por el equipo docente de la materia. Hicimos una implementación alternativa de tres funciones involucradas en la simulación en lenguaje ensamblador x86-64 utilizando instrucciones de procesamiento vectorial. El estudio comparativo consiste en el estudio de la performance de las implementaciones en sendos lenguajes.

La presente investigación es importante porque a pesar de que la velocidad de procesamiento y la velocidad de acceso a memoria aumentan a lo largo de los años, la curva que describe el incremento de la primera tiene un mayor orden de magnitud que la curva de la segunda. Esto significa que la brecha es cada vez mayor y se traduce en un costo porcentual creciente de los accesos a memoria sobre el costo total de procesamiento de un programa.

Asimismo, esta investigación de caracter educativo es importante para programadores que se desempeñan en la academia o en la industria porque motiva a la optimización del código, a la medición de performance de los programas y a la profundización del conocimiento de lo que sucede en el procesador en la ejecución de programas escritos en lenguajes de alto nivel.

2. Desarrollo

El código fuente de la simulación está escrito en lenguaje C y el compilador utilizado es GCC. Para las funciones de C *solver_lin_solve*, *solver_set_bnd* y *solver_project* hicimos implementaciones alternativas escritas en el lenguaje ensamblador de la familia de procesadores Intel x86-64. Las instrucciones vectoriales en lenguaje ensamblador utilizan registros de 128 bits. Los estados de la simulación se representan mediante matrices de números decimales de punto flotante de precisión simple (32 bits). Los algoritmos asumen que todas las matrices en una ejecución particular constan de $n + 2$ filas y $n + 2$ columnas con $n \geq 4$ y n múltiplo de 4. A continuación se explica la implementación en ensamblador de las tres funciones.

2.1. Función solver_lin_solve

El ciclo principal de la función está formado por dos ciclos: un ciclo que recorre las columnas de las matrices y otro ciclo que recorre las filas. A causa de que las matrices contienen dimensiones múltiplos de cuatro, sin contar con las filas y columnas borde que no se usan en esta función, y los elementos de las matrices ocupan 4 bytes hemos decidido fetchear, fetchear es cargar de memoria, de a cuatro elementos consecutivos de una fila para aprovechar el espacio de los registros xmm, que es de 16 bytes, y por lo tanto podemos cargar cuatro elementos en un solo fetch. Así el índice de las columnas, i, queda en un rango entre 1 y $N/4$, donde cada incremento de i representa avance de a cuatro columnas. El ciclo sobre columnas es el primero. Adentro tenemos otro ciclo donde se recorren las filas de a una.

Empezamos cargando desde matriz x. En cada iteración sobre fila se chequea si la fila es la primera, tal que si es así entonces se realiza $1er$ fetch de cuatro elementos consecutivos desde matriz x en un xmm llamado xmm_piso. Si no es la primera entonces se copian de un registro xmm, llamado xmm_backup, que contiene los cuatro resultados de ciclo anterior. Luego realizamos $2do$ fetch de a cuatro de una fila siguiente, en un xmm llamado xmm_left y avanzamos dos posiciones para un $3er$ fetch de a cuatro en un xmm llamado xmm_right. En la siguiente fila fetcheamos de a cuatro en un xmm llamado xmm_techo. Los elementos cargados desde x forman un bloque de elementos (ver figura 1 (a)) de donde vamos a obtener cuatro resultados a partir de seis accesos a memoria. Por otra parte de matriz $x0$ fetcheamos cuatro elementos consecutivos en un xmm llamado xmm_x0.

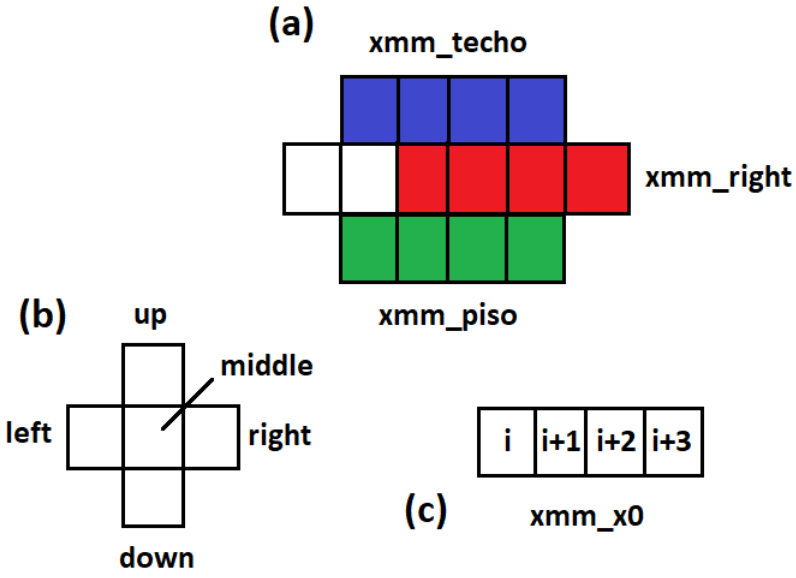

Figura 1: (a) Bloque de elementos de matriz x. (b) Sumandos de x en código c. (c) Empacado de elementos de $x0$.

Representamos en figura 1 (b) los sumandos de matriz x que aparecen en cuerpo de ciclo principal en código c (ver código $solver\_lin\_solve$ en $solver.c$): up representa al sumando $x[IX(i, j+1)]$, down a $x[IX(i, j-1)]$, left a $x[IX(i-1, j)$ y right a $x[IX(i+1, j)]$. Por último middle representa a sumando $x0[IX(i, j)]$. En bloque de figura 1 (a) vemos que xmm_techo empaca a los cuatro sumandos up, xmm_piso empaca a los cuatro down y xmm_right empaca a los cuatro right de los cuatro puntos centrales del bloque. También vemos, en figura 1 (c), que xmm_x0 empaca a los cuatro middle asociados a los puntos centrales del bloque. La idea que usamos para paralelizar cálculos es sumar los valores de los registros xmm_piso, xmm_techo y xmm_right en paralelo obteniendo cuatro sumas parciales. Luego completar la suma alrededor de un punto individualmente, a causa de dependencia de vecino izquierdo, sumando up, down, left y right, comenzando con entorno de primero de los cuatro puntos centrales. Terminar las operaciones sobre ese punto, usando primer middle empaquetado en xmm_x0, y pasar a operar sobre vecino derecho usando como sumando left de este al resultado obtenido. De esta forma se respeta el código c en donde sobre una fila de x los puntos se actualizan hacia derecha, es decir que dependen de sus vecinos izquierdos.

Volviendo al cuerpo del ciclo, una vez que tenemos los datos desempaquetamos con instrucción $cvtps2pd$, que convierte los single en parte baja de xmm fuente a double en xmm destino. Luego shifteamos a derecha con $psrldq$ para acceder a los single de parte alta de los xmm y convertirlos a

double. En este paso obtenemos *xmm_piso_low* y *xmm_piso_high* que corresponden a desempacar a double los 2 single en parte baja y los 2 single en parte alta de *xmm_piso* respectivamente. Repetimos este procedimiento con *xmm_right* obteniendo *xmm_right_low* y *xmm_right_high*. De misma manera desempaquetamos *xmm_techo* en *xmm_techo_low* y *xmm_techo_high*. Por otra parte se convierten los singles *a* y *c* a double en *xmm_a* y *xmm_b* con instrucción *cvtss2sd*, que convierte escalar single de parte menos significativa de *xmm* fuente a escalar double en parte baja de *xmm* destino. Por último convertimos los single de *xmm_x0* usando instrucción *cvtps2pd* en double obteniendo dos *xmm*: *xmm_x0_low* y *xmm_x0_high*. También convertimos primer single en xmm_left a double con instrucción *cvtss2sd* que convierte single de parte menos significativa de *xmm* fuente a double en parte baja de *xmm* destino. Se guarda esto en xmm_res, que usaremos como contenedor de resultados finales.

Siguiente paso se suman los dos double de *xmm_techo_low* y *xmm_right_low* guardándose en un *xmm* temporal. Luego a estos dos doubles resultado de suma entre techo y lado derecho les sumamos los dos doubles en *xmm_piso_low* obteniendo sumas parciales $x[IX(i+1,j)] + x[IX(i,j-1)] + x[IX(i,j+1)]$ de entorno de dos puntos de x. Guardamos esto en *xmm* llamado *xmm_sum_parcial_low* y repetimos procedimiento con los *xmm* de partes altas: *xmm_techo_high*, *xmm_piso_high* y *xmm_right_high*. Guardamos estos resultados en *xmm_sum_parcial_high* asociados a otros dos puntos de x.

Paso siguiente entramos en ciclo que se repite cuatro veces operando sobre doubles de parte baja y obteniendo en cada iteración un resultado a guardar en matriz x. Primero se suman escalar de *xmm_res* con double en parte baja de alguno de los *xmm_sum_parcial* con instrucción *addsd* que suma escalares doubles, obteniéndose $x[IX(i-1,j)]+x[IX(i+1,j)]+x[IX(i,j-1)]+x[IX(i,j+1)]$, y guardándose en *xmm_res*. Entonces se multiplica con escalar *xmm_a* usando instrucción *mulsd*, que multiplica escalares double, obteniéndose $a*(x[IX(i-1,j)]+x[IX(i+1,j)]+x[IX(i,j-1)]+x[IX(i,j+1)])$ y lo guardamos en *xmm_res*. Luego le sumamos double en parte baja de *xmm_x0* obteniendo $x0[IX(i,j)] + a*(x[IX(i-1,j)] + x[IX(i+1,j)] + x[IX(i,j-1)] + x[IX(i,j+1)])$ y lo guardamos en *xmm_res*. Paso seguido dividimos *xmm_res* por escalar *xmm_c* con instrucción *divsd*, que divide escalares double, obteniendo $(x0[IX(i,j)]+a*(x[IX(i-1,j)]+x[IX(i+1,j)]+x[IX(i,j-1)] + x[IX(i,j+1)]))/c$ y lo guardamos en *xmm_res*. En primer iteración *xmm_res* contiene sumando left de $1er$ punto central de bloque en figura 1 y al sumar con escalar en parte baja de *xmm_sum_parcial_low* obtenemos suma de up, low, right y left para ese punto. Realizamos las operaciones restantes y guardamos este resultado en *xmm_res* para ser usado como sumando left en siguiente iteración. Finalmente convertimos el resultado en *xmm_res* de double a single con instrucción *cvtsd2ss* y lo movemos a *xmm_backup* con instrucción *movss*, que mueve un escalar single a parte baja de *xmm* destino. Una vez guardado resultado en *xmm_backup* shifteamos este registro a izquierda con instrucción *pslldq*, que mueve a izquierda una double quadword, moviendo de a 4 bytes el valor recientemente cargado. De esta manera dejamos espacio para un single en parte baja de *xmm_backup* que recibirá a próximo resultado desde *xmm_res*. En cada iteración de este ciclo shifteamos los registros *xmm_sum_parcial_low* y *xmm_x0_low* a derecha, con *psrldq*, para acceder a las partes altas de estos y, en caso de haber usado los dos doubles de cada registro pasamos a operar con *xmm_sum_parcial_high* y *xmm_x0_high*.

Luego de salir de este ciclo de cuatro iteraciones tenemos en *xmm_backup* los cuatro resultados a subir en matriz x pero con las posiciones invertidas por como se cargaron en las iteraciones. Entonces intercambiamos sus posiciones con instrucción $pshufd$ que reubica doublewords en destino con ayuda de un registro temporal. Una vez hecho esto guardamos estos cuatro resultados en matriz x, con instrucción *movups*, y saltamos a siguiente fila terminando una iteración de ciclo sobre filas. En próxima iteración no tenemos que cargar de memoria los cuatro resultados obtenidos, ya que los tenemos guardados en *xmm_backup*, que sobreescribieron fila de x y son requeridos. Al iterar sobre N filas terminamos una iteración de ciclo principal y avanzamos a las siguientes cuatro columnas de x y $x0$. Una vez iterado $N/4$ veces ciclo principal salimos de este con bloque de $N \times N$ de x actualizado.

2.2. Función solver_set_bnd

La función solver_set_bnd se encarga de actualizar los valores del borde. El algoritmo consta de tres partes: el procesamiento de los bordes horizontales (la primera y la última fila de la matriz), el procesamiento de los bordes verticales (la primera y la última columa) y el procesamiento de las esquinas. Los primeros dos se realizan en un ciclo.

El procesamiento horizontal consiste en sobreescribir las celdas de las filas 0 y $n + 1$ con las filas 1 y n respectivamente. En algunos casos el valor de la celda se invierte antes de la escritura, según los datos de entrada (para más información consulte el código fuente). La lectura, escritura y cambio de signo se realiza de a 4 elementos con un registro vectorial, ya que las celdas de cada fila se encuentran contiguas en memoria. El procesamiento excluye a la primera y última celda de las filas involucradas.

El procesamiento vertical es similar al horizontal porque también consiste en sobreescribir respectivamente las columnas 0 y $n + 1$ con las columnas 1 y n excluyendo la primera y la última celda de las columnas. Como sucede en el procesamiento de las filas, en algunos casos el valor de la celda se invierte antes de la escritura, según los datos de entrada. La diferencia es que solo el cambio de signo se realiza de a 4 elementos con un registro vectorial; la lectura y escritura se realizan individualmente porque las celdas no se encuentran contiguas en memoria.

El procesamiento de las esquinas consiste en sumar el valor de las dos celdas contiguas a cada esquina (contiguas en la interpretación matricial, no en memoria), luego dividir por dos ese valor y escribirlo en la esquina más cercana a esas celdas. El resultado de este cálculo está determinado por la operación realizada anteriormente sobre las filas y columnas, por lo tanto podemos evitar los cálculos en pos de una mayor performance si consideramos dos casos, a saber: si hubo o no cambios de signo. El cambio de signo, como ya fue mencionado anteriormente, está determinado por los parámetros de entrada.

El primer caso se da cuando hay cambio de signo de algunos valores. El cambio de signo respeta la siguiente regla: si el signo de cada elemento de las filas fue cambiado, entonces el signo de cada celda de las columnas no cambió, y viceversa. En este caso el resultado de la cuenta da siempre cero, por lo cual el proceso consiste en la escritura de este valor en las cuatro esquinas.

En el segundo caso no hay cambio de signo de ninguna celda. En este caso las esquinas serán sobreescritas con el valor de cualquiera de las dos celdas contiguas *nodiagonales* a ella, ya que ambas tendrán el mismo valor (y el resultado de la cuenta es este valor).

2.3. Función solver_project

El algoritmo con el que implementamos esta función se lleva adelande en varias etapas, por un lado los calculos que se hacen dentro del mismo algoritmo, y por otro, los que estan externalizados y llevados a cabo por las otras funciones que fueron implementadas en assembler en este trabajo.

Esta implementación sigue la linea del código de la misma función escrito en C, con dos ciclos y llamadas a otras funciones. Como dentro de la función llamamos a solver_lin_solve y solver_set_bnd y estos terminan alterando los parametros que utilizamos, necesitamos hacer esos procesamientos en distintos momentos dentro de la ejecución y de ahí la necesidad de hacerlo en dos ciclos.

Los calculos que hacemos dentro de la función, trabajan sobre la matriz *div*, con las matrices u y v de solver. Para cada celda de la matriz *div*, vamos a operar con las celdas aledañas (por

arriba y por abajo, no las que están en diagonal) a la de la misma posicion de las matrices u y v (tengamos en cuenta que todas estas son matrices de igual dimensión). Notemos que solo podemos realizar estas operaciónes en las celdas de la matriz que no se encuentran en los bordes de la misma. Entonces, vamos a querer hacer estas operaciones con instruciones SIMD.

El mayor problema acá se presentó cuando quisimos hacer las operaciones que tomaban las celdas de distintas filas de la matriz. Esto lo resolvimos teniendo 4 punteros, uno a cada celda *aledaña* con la que ibamos a operar, empezando por el primer elemento de nuestra submatriz. Es decir, para el (1,1) tenemos un puntero al (0,1), otro al (2,1), otro al (1,0) y otro al (1,2). Con esto podemos usar SIMD tomando de a 4 elementos, tanto en la matriz que vamos a modificar com en las otras con las que vamos a operar, donde ahora tenemos punteros a los elementos que nos interesan.

Cuando tomamos la matriz *div* (la "principal", o sobre la cual estamos operando), con una instrucción SIMD avanzamos de a 4 celdas. Como la submatriz en la que nos movemos tiene el tamaño de sus filas multiplo de 4 (dato del enunciado), no tenemos que preocuparnos por tomar algún elemento no deseado cuando la función termine de recorrer una fila.

Con lo que sí tenemos que tener cuidado es con cómo pasar de una fila a otra, por que en la estructura, entre una fila y otra de la submatriz en la que trabajamos tenemos 2 elementos. Para resolver esto, chequeamos con el tamaño de cada fila (dato) si la terminamos de recorrer.

En caso de no haber terminado, seguimos operando sobre la misma fila, y en caso de haber terminado de recorrerla, lo que hacemos es sumar dos posiciónes extras para la próxima iteración, salteandonos así los bordes de la matriz, que no queremos modificar ahora.

Ahora lo único que nos restaba hacer en cada ciclo era dejar en cero todas las celdas de la misma submatriz pero de p (o sea, la matriz p sin sus bordes). En nuestra implementación lo hicimos al principio del ciclo (aun que podríamos haberlo hecho en otro momento dentro del ciclo, al no utilizar la matriz p para nada más a lo largo de ese primer ciclo). Esta matriz la recorremos en conjunto con las demás, al ser exactamente del mismo tamaño de las demás.

Con esto termina el primer ciclo de esta función. Ahora llamamos dos veces a solver_set_bnd para completar los bordes de la matriz, que no los había tocado anteriormente. La llamamos una vez con la matriz *div* y otra con la matriz p. Luego, llamamos a solver_lin_solve.

Como podemos observar, estas llamadas a otras funciones modifican los parametros con los que trabajamos. Ahora en el segundo ciclo, trabajamos sobre las matrices u y v de solver (ambas de tamaño (N+1) x (N+1)), y al igual que en el ciclo anterior lo que hacemos es recorrer la submatriz de N x N formada por la matriz sin sus bordes. En este caso lo único que cambia respecto del ciclo anterior es que son otras operaciones las que hay que hacer pero no son muy distintas, dado que volvemos a usar esas celdas aledañas. Luego vamos a recorrer las matrices de manera análoga a como hicimos en el primer ciclo.

Y por último, llamamos a solver_set_bnd dos veces, como hicimos antes, pero en este caso con las matrices u y v.

2.4. Eficiencia C vs paralelismo simd ASM

Nuestra hipótesis es que código ASM con instrucciones simd aprovecha el uso de vectores más eficientemente que código C. Para averiguar esto hemos planteado experimento en que seleccionamos tamaño de matriz, en atributos de estructura solver, y testeamos los tiempos en ticks que tarda la ejecución de cada código. Los tamaños elegidos son los que usa la cátedra para la demo 2, es decir tamaños de 16x16, 32x32, 64x64, 128x128, 256x256 y 512x512. Se repiten 100 veces la ejecución de cada código, C y ASM, para cada tamaño y luego se promedian evitando los outliers. Entonces se comparan los promedios y se sacan conclusiones.

3. Resultados

3.1. Función solver set bnd

Se evaluó el código en assembler, llamado ASM, y código C de la función solver set bound sobre seis tamaños distintos de matrices. Al variar parámetro b, valores de 1, 2, 3 y 10, se obtienen promedios similares en caso de matriz con tamaño 512x512 (ver tabla), sacando outliers. Los resultados se mantienen alrededor de 18500 ticks para ASM y 54000 ticks para C. Repetimos escenario con matriz de tamaño 16x16, variando b en mismos valores que para tamaño 512x512. Aunque se nota variación de tiempos entre mediciones (ver cuadro 1) no se nota gran cambio en los resultados, manteniendose el promedio de tiempos C alrededor de 2000 ticks y tiempos ASM alrededor de 350 ticks.

A causa de esto y de que no queremos llenar de gráficas innecesarias hemos decidido fijar el

	C	ASM
*solver_set_bnd*_100_16_*b*1	2061.989474	335.134021
*solver_set_bnd*_100_16_*b*2	2081.364583	360.896907
*solver_set_bnd*_100_16_*b*3	2068.020408	376.135417
*solver_set_bnd*_100_512_*b*1	56134.061856	18529.938144
*solver_set_bnd*_100_512_*b*2	54054.041667	19752.25000
*solver_set_bnd*_100_512_*b*3	54835.708333	18544.926316

Cuadro 1: Tabla de promedios ticks función solver set bnd para tamaños 16x16 y 512x512.

parámetro b en 2 para restantes experimentaciones de función. Se muestra en figura 2 promedio de ticks gastado en ejecución de los códigos, donde para cada tamaño los códigos se ejecutaron 100 veces.

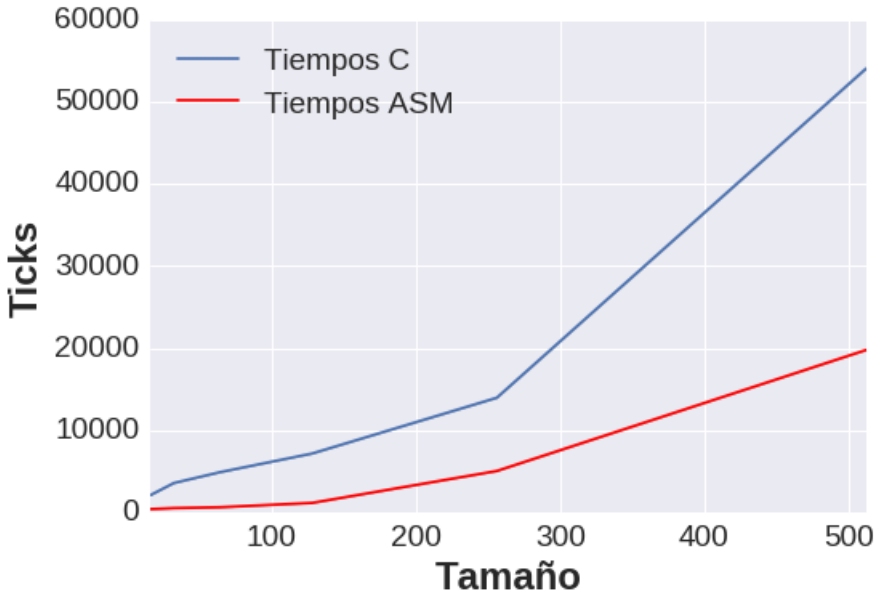

Figura 2: Tiempos en ticks de ejecución de código C vs código ASM para función solver set bnd

Se ve que el código C gasta más tiempo que código ASM, agrandandose esta diferencia a medida que aumenta el tamaño de matriz.

3.2. Función solver lin solve

En este caso hemos variado los parámetros a, b y c para tamaños de matriz 16x16 y 512x512. En la tabla siguiente se muestran los promedios obtenidos sobre datos sin outliers para valores de: $a = 1{,}0f$ y $c = 4{,}0f$, llamado $1erOp\_a\_y\_c$, $a = 0{,}3f$ y $c = 2{,}8f$, llamado $2daOp\_a\_y\_c$, y $a = -10{,}0f$, $c = 0{,}02f$, llamado $3raOp\_a\_y\_c$. Los valores de b son: 1, 2, 3 y 10. En el caso de tamaño 16x16 obtenemos que el promedio de tiempos de ejecución sin outliers para código en C está alrededor de los 400000 ticks y para código ASM está alrededor de 300000 ticks. En el caso de tamaño 512x5122 se obtiene que promedio de tiempos C está alrededor de $2{,}3 * e + 08$ y tiempos ASM está alrededor de $1{,}6 * e + 08$ ticks.
Se ve en la cuadro 2 que se repite una proporción de casi el doble de gasto temporal en ejecuciones

	C	ASM
$solver\_lin\_solve\_b1\_1erOp\_a\_y\_c\_16$	392938.224490	236550.530612
$solver\_lin\_solve\_b2\_1erOp\_a\_y\_c\_16$	442471.040816	252094.406250
$solver\_lin\_solve\_b3\_1erOp\_a\_y\_c\_16$	429083.391753	253708.344086
$solver\_lin\_solve\_b10\_1erOp\_a\_y\_c\_16$	436142.561224	261522.363636
$solver\_lin\_solve\_b1\_2daOp\_a\_y\_c\_16$	461250.443299	369954.084211
$solver\_lin\_solve\_b10\_2daOp\_a\_y\_c\_16$	430715.626263	352677.979592
$solver\_lin\_solve\_b1\_3raOp\_a\_y\_c\_16$	427202.773196	262558.585859
$solver\_lin\_solve\_b10\_3raOp\_a\_y\_c\_16$	429630.377551	237256.265306
$solver\_lin\_solve\_b1\_1raOp\_a\_y\_c\_512$	2.320231e+08	1.576663e+08
$solver\_lin\_solve\_b2\_1raOp\_a\_y\_c\_512$	2.364991e+08	1.582853e+08
$solver\_lin\_solve\_b10\_1raOp\_a\_y\_c\_512$	2.313464e+08	1.570360e+08
$solver\_lin\_solve\_b1\_3raOp\_a\_y\_c\_512$	2.343117e+08	1.585373e+08
$solver\_lin\_solve\_b10\_3raOp\_a\_y\_c\_512$	2.337076e+08	1.611139e+08

Cuadro 2: Tabla de promedios ticks función solver lin solve para tamaños 16x16 y 512x512.

C respecto a ejecuciones ASM. Entonces decidimos elegir parámetros de $1erOp\_a\_y\_c$, $b = 2$ y las matrices $solver \to u$, $solver \to v$. Con estos parámetros variamos el tamaño de las matrices y graficamos los tiempos (ver figura 3).
Se observa notable diferencia de gasto temporal, código C gasta más ticks que ASM.

3.3. Función solver project

Hemos medido la función sobre cuatro matrices diferentes: $1erOp$ que comienza con valor (0.1,0.2) en posición (0,0) y luego a medida que avanzamos en posiciones se incrementa en uno el valor en posición anterior y se asigna ese resultado, $2daOp$ con mismo proceso pero comenzando en (0.09,-100), $3eraOp$ comenzando en (-10,0.08) y $4taOp$ comenzando en (1000,2000). Se evalúan matrices con tamaño 16x16 y 512x512 para observar si hay gran cambio en las proporciones de tiempo al ejecutar código. Se observa en cuadro 3 los resultados y se ve que para ambos tamaños el código C gasta alrededor de un tercio más que tiempo gastado por código ASM.

A causa de que al variar la matriz obtenemos resultados similares, alrededor de 380000 ticks en código C y alrededor de 270000 en asm para tamaño 16x16 (ver tabla), hemos decidido evaluar los códigos sobre matriz $1erOp$. En la gráfica se muestran los promedios para distintos tamaños (figura 4).

Observando la gráfica vemos que en este caso la diferencia de tiempos no es tan grande. Notamos que el código de la función hace repetidamente llamadas a las otras funciones: solver set bnd y solver lin solve. Se ve que a medida que el tamaño de matriz aumenta las curvas que representan los gastos de tiempo tienden a separarse.

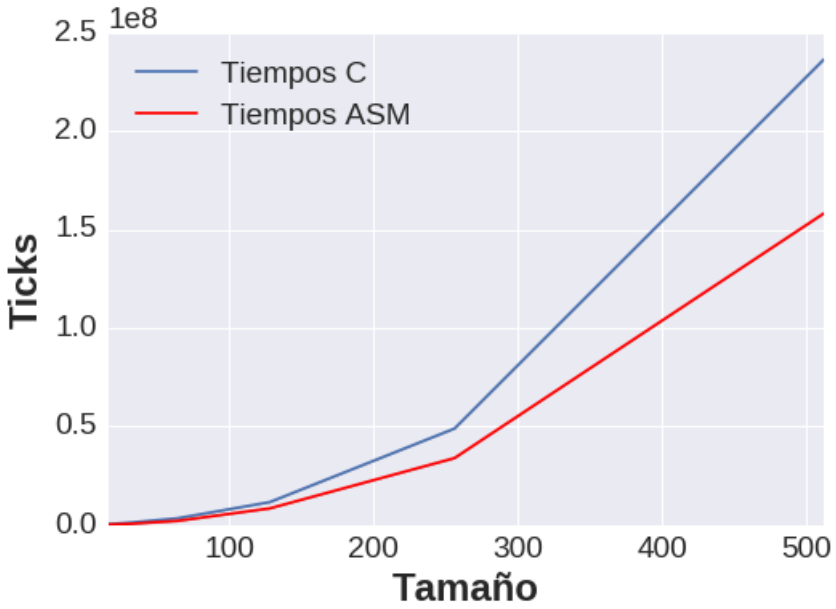

Figura 3: Tiempos en ticks de ejecución de código C vs código ASM para función solver lin solve

	C	SD(C)	ASM	SD(ASM)
solver_project_1erOp_matrices_16	384948.313131	36503.785424	271883.232323	28601.904704
solver_project_2daOp_matrices_16	385072.787879	34959.635332	271078.418367	15269.103250
solver_project_3eraOp_matrices_16	382180.464646	29999.207094	273869.231579	13431.150139
solver_project_4taOp_matrices_16	353107.683673	24055.789775	259020.680412	12374.032677
solver_project_1erOp_matrices_512	1.925452e+08	1.804747e+06	1.584625e+08	1.589767e+06
solver_project_2daOp_matrices_512	1.939796e+08	3.205834e+06	1.591681e+08	2.147394e+06
solver_project_4taOp_matrices_512	1.938470e+08	2.764452e+06	1.670031e+08	8.623912e+06

Cuadro 3: Tabla de promedios ticks función solver project para tamaños 16x16 y 512x512. SD es desvío estandar.

4. Conclusión

4.1. Implementación en código ASM

En implementación de función solver_lin solve en lenguaje ASM se aprovechó la capacidad de los registros *xmm* de manera que con cinco accesos a memoria obtenemos resultados de 4 puntos consecutivos, suponemos que el código C hace más accesos a memoria mirando su implementación. Desempaquetando datos de single a double float no notamos mejora en resultados esperados (comparación hecha con resultados de cátedra). Obtuvimos precisión de $10 * e - 4$ en los resultados respecto a los resultados de la cátedra.

4.2. Eficiencia C vs paralelismo simd ASM

Al comparar implementaciones en C y ASM de las funciones solver set bnd, solver_lin_solve y solver_project obtuvimos que variando los parámetros se repetía una proporción de gasto temporal a favor de implementación ASM, menos gasto, y en contra de implementación C, más gasto temporal con código C.

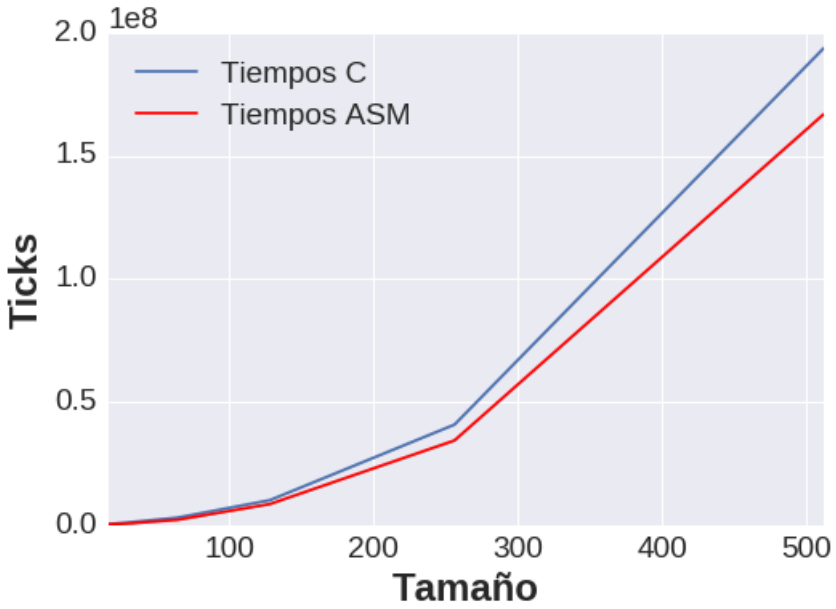

Figura 4: Tiempos en ticks de ejecución de código C vs código ASM para función solver project

En la función solver_set_bnd se observó gran diferencia en gastos temporales, la implementación C gastó alrededor de cinco veces el tiempo gastado por implementación ASM. Suponemos esto a causa de que la función llamadas a otras funciones.

En la función solver_lin_solve se observó marcada diferencia en tiempo de ejecución, código C gastó alrededor del doble de tiempo que código ASM. Suponemos que la llamada a otra función influencia en la disminución de diferencia de tiempos, menor diferencia comparada a lo obtenido en función solver set bnd.

En función solver_project se observó poca diferencia en tiempos de ejecución, implementación C gastó alrededor de un tercio de tiempo que código ASM. Suponemos esto debido a las repetidas llamadas a otras funciones que se hace en el código.

En general se observa ventaja temporal usando implementación ASM.

Queda como pendiente analizar tiempos de ejecución optimizando el compilador con opciones ($O1, 02$ y 03).