



**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

9 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

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 segunda¹. 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 carácter 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_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 columna) y el procesamiento de las esquinas. Los primeros dos se realizan en un ciclo.

¹<http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf>

El procesamiento horizontal consiste en sobrescribir respectivamente las celdas de las filas 0 y $n + 1$ con las filas 1 y n excluyendo la primera y la última celda de las filas. Si el valor del parámetro b es igual a 2 se cambia el signo del valor a escribir por su contrario, en otro caso se escribe el valor original. La lectura, escritura y cambio de signo se realizan de a 4 elementos con un registro vectorial, ya que las celdas de cada fila se encuentran contiguas en memoria.

El procesamiento vertical consiste en sobrescribir respectivamente las columnas 0 y $n + 1$ con las columnas 1 y n excluyendo la primera y la última celda de las columnas. Si el valor del parámetro b es igual a 1 se cambia el signo del valor a escribir por su contrario, en otro caso se escribe el valor original. El cambio de signo se realiza de a 4 elementos con un registro vectorial, pero 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. La programación en lenguaje ensamblador implementa un procesamiento alternativo que produce el mismo resultado utilizando instrucciones de asignación (*mov*) y operaciones con enteros en lugar de la operación de punto flotante, que es más costosa y puede generar un error de redondeo. Consideraremos dos casos.

El primer caso se da cuando el parámetro b es igual a 1 o 2. Si b es 1 el signo de los valores de los bordes verticales es el opuesto al valor de las celdas de las columnas de origen pero los valores de los bordes horizontales mantienen el signo original. Si b es 2 se invierte el signo los bordes horizontales y se mantiene el signo de los bordes verticales. Por lo tanto, el proceso consiste en escribir cero en las cuatro esquinas, ya que los dos valores adyacentes a cada una son iguales en módulo y con signos opuestos.

El segundo caso se da cuando b tiene otro valor. El proceso consiste en sobrescribir cada esquina con el valor de cualquiera de las dos celdas adyacentes, ya que al no haber cambios de signo estas celdas tienen el mismo valor.

2.2. 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 fetchearnos 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 $x0$ fetchearnos cuatro elementos consecutivos en un *xmm*

llamado *xmm_x0*.

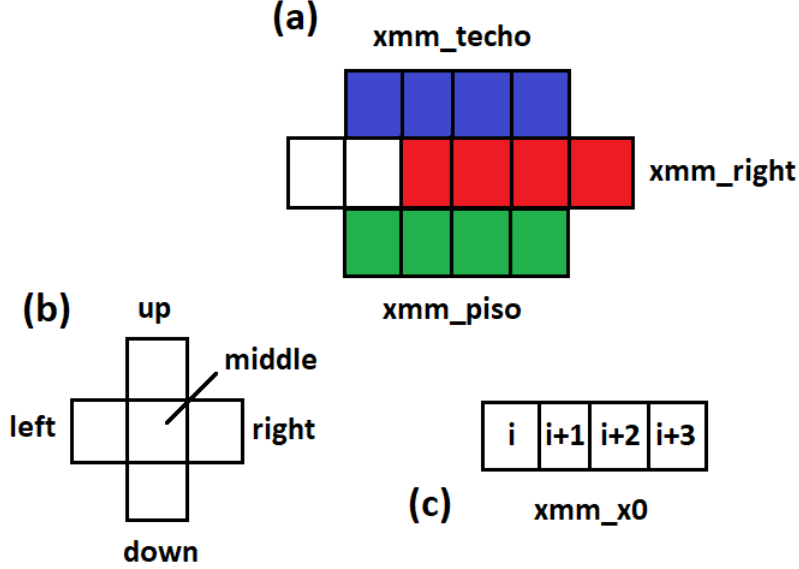


Figura 1: (a) Bloque de elementos de matriz x . (b) Sumandos de x en código c . (c) Empacado de elementos de x_0 .

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 $x_0[IX(i, j)]$. En bloque de figura 1 (a) vemos que *xmm_techo* empaqueta a los cuatro sumandos *up*, *xmm_piso* empaqueta a los cuatro *down* y *xmm_right* empaqueta a los cuatro *right* de los cuatro puntos centrales del bloque. También vemos, en figura 1 (c), que *xmm_x0* empaqueta 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 *cvts2sd*, 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 *cvts2sd* 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 *cvtss2ss* 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 sobrescribieron 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.

2.3. Función solver_project

El algoritmo con el que implementamos esta función se lleva adelante 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 posición de las matrices *u* y *v* (tengamos en cuenta que todas estas son matrices de igual dimensión). Notemos que solo podemos realizar estas operaciones en las celdas de la matriz que no se encuentran en los bordes de la misma. Entonces, vamos a querer hacer estas operaciones con instrucciones 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 íbamos 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 como 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 múltiplo 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 posiciones extras para la próxima iteración, saltandonos 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 recorreremos 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 $(N+1) \times (N+1)$), y al igual que en el ciclo anterior lo que hacemos es recorrer la submatriz de $N \times N$ 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.

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

3. Resultados

3.1. Función Solver Set Bnd

3.1.1. Compilación hecha con gcc en opción o0

Se evaluó el código en assembler con instrucciones SIMD, llamado ASM, y código C de la función Solver Set Bnd sobre seis tamaños distintos de matrices: 16x16, 32x32, 64x64, 128x128, 256x256 y 512x512. Empezamos restringiéndonos a tamaños 16x16 y 512x512 de manera que si observamos tendencia evitaremos evaluar todos los tamaños con todas las variaciones de parámetros. Al variar parámetro b , valores de 1, 2, 3 y 10, se obtienen promedios similares en caso de matriz con tamaño 512x512 (ver Cuadro 1), alrededor de 50000 ticks para código C, sacando outliers. En caso de implementación ASM se siente levemente mayor gasto con b mayor a 2 y menor gasto con b igual a 1 y 2, aunque se mantiene alrededor de los 16000 ticks, menor a los de C. El desvío estandar en caso C supera los 3000 ticks y para ASM supera los 1000 ticks, levemente dispersos alrededor de la media en ambos casos. Con esto, y el porcentaje de datos con el que se promedió (columna %), tomamos al promedio como representante de la mayoría de las muestras.

Repetimos escenario con matriz de tamaño 16x16, variando b en mismos valores que para tamaño 512x512. Aunque se nota variación de tiempos entre mediciones no se nota gran cambio en los resultados, manteniendose el promedio de tiempos C alrededor de 1800 ticks, con desvío estandar arriba de 100 ticks, tiempos ASM alrededor de 350 ticks, con leve variación, igual que para caso 512x512 a causa de distintas variantes de b , y desvío estandar alrededor de 50 ticks para ASM. Se observa en caso de 16x16 que para distintos valores de b la proporción de gasto temporal de código C se mantiene en seis veces el gasto que tiene código ASM. A causa de esta tendencia decidimos fijar b en 1 y evaluar en todos los tamaños propuestos a la función.

	C	%	SD(C)	ASM	%	SD(ASM)
<i>solver_set_bnd_100_16_b1</i>	1728.802	96	176.465	308.638	94	51.883
<i>solver_set_bnd_100_16_b2</i>	1766.234	98	203.315	313.909	88	13.749
<i>solver_set_bnd_100_16_b3</i>	1838.315	92	133.411	388.340	94	66.619
<i>solver_set_bnd_100_16_b4</i>	1746.914	94	115.583	399.156	96	78.437
<i>solver_set_bnd_100_512_b1</i>	50553.760	75	7963.533	15610.465	73	1270.874
<i>solver_set_bnd_100_512_b2</i>	45526.361	83	3697.485	15732.926	82	1622.451
<i>solver_set_bnd_100_512_b3</i>	51947.550	80	4330.216	17793.384	78	1225.667
<i>solver_set_bnd_100_512_b4</i>	49148.481	83	3601.135	17413.631	76	1530.499

Cuadro 1: Tabla de promedios de función Solver Set Bnd para tamaños 16x16 y 512x512. Columna % es porcentaje de datos (no outlier) con que se promedió. SD es desviación estandar.

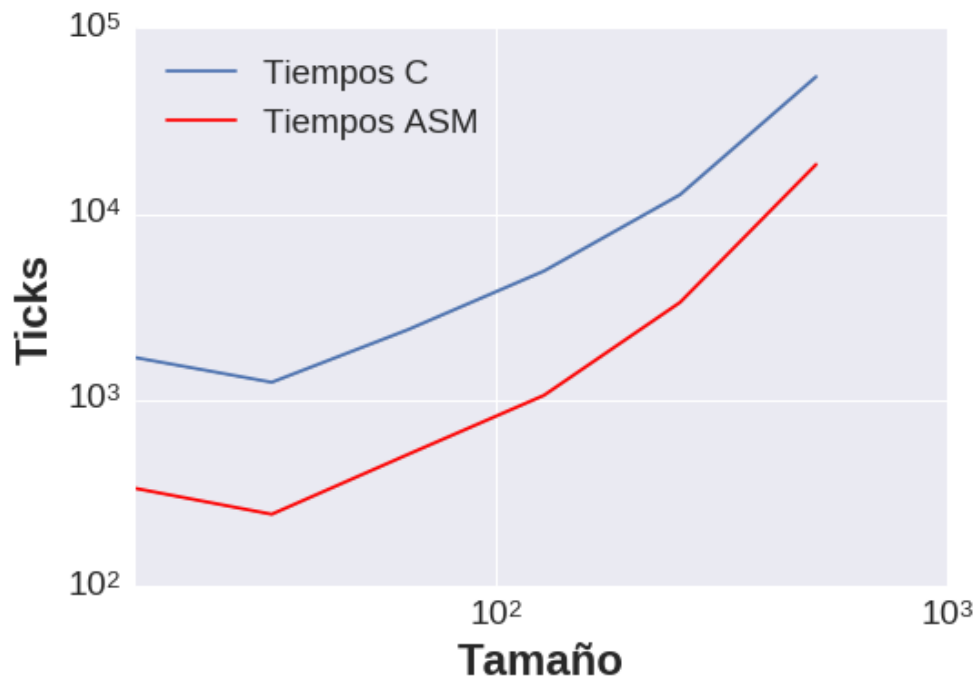
Se muestra en Figura 2(a) promedio de ticks gastado en ejecución de los códigos, donde para cada tamaño los códigos se ejecutaron 100 veces.

Se ve que el código C gasta más tiempo que código ASM, achicándose esta diferencia a medida que aumenta el tamaño de matriz. Suponemos que esta ventaja de ASM sobre C se debe al uso de instrucciones SIMD en código ASM.

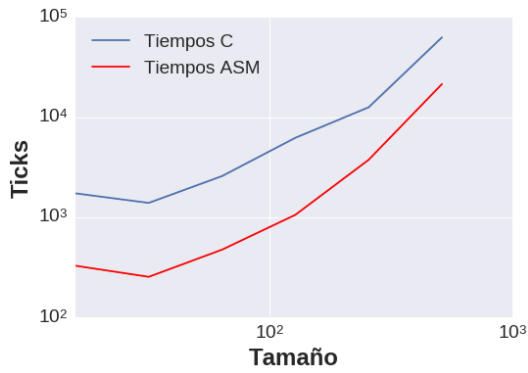
3.1.2. Compilación hecha con gcc en opción o1

Repetimos parámetros de función (b en 1) para poder comparar las gráficas. El resultado se observa en Figura 2(b).

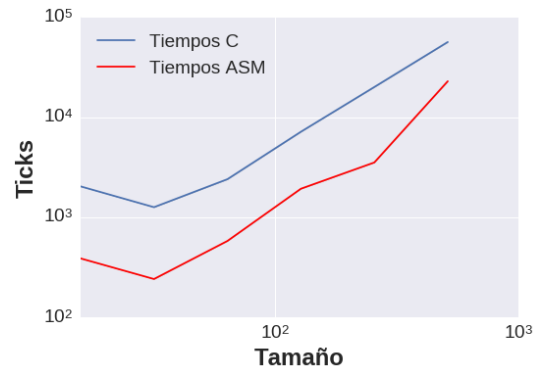
No se nota mejora en tiempos C, ambas gráficas, la de Solver Set Bnd y gcc o0 junto a la de Solver Set Bnd y gcc o1, muestran comportamientos similares. Tal vez observe algo en siguiente opción de optimización.



(a) Compilación hecha con gcc en opción o0



(b) Compilación hecha con gcc en opción o1



(c) Compilación hecha con gcc en opción o3

Figura 2: Tiempos en ticks de ejecución de código C vs código ASM para función Solver Set Bnd.

3.1.3. Compilación hecha con gcc en opción o3

Usamos mismas opciones de b que para anteriores mediciones, b en 1. El resultado se muestra en Figura 2(c).

No vemos mejora en tiempos C, variando apenas los tiempos de este código a favor y en contra.

3.2. Función Solver Lin Solve

3.2.1. Compilación hecha con gcc en opción o0

En este caso hemos variado los parámetros a , b y c solamente para tamaños de matriz 16x16 y 512x512. En la tabla siguiente se muestran los promedios obtenidos sobre datos sin outliers para valores de: $a = 1,0$, $b = 1$ y $c = 4,0$, llamado *1erOp*, $a = 0,3$, $b = 2$ y $c = 2,8$, llamado *2daOp*, $a = 100,0$, $b = 3$, $c = 20,0$ llamado *3raOp*, $a = -10,0$, $b = 10$, $c = 0,02$, llamado *4taOp*. Observamos los resultados en Cuadro 2. Para tamaño 16x16 observamos que se destaca el caso *1erOp*, donde porcentaje de datos no outliers es del 49 % pero observando la gráfica de los datos (no incluida) vemos que el 61 % restante se reparte un 30 % arriba y otro 30 % abajo de los no outliers, y por lo tanto este 49 % refleja el comportamiento de la mayoría de los datos. Por otra parte en caso *2daOp* se observa pobre ventaja de código ASM, de alrededor del 10 %, sobre código C, mientras que en los otros casos se obtiene un porcentaje de ventaja a favor de ASM levemente mayor. Observando el desvío estandar, datos poco dispersos respecto a la media, y el porcentaje de datos promediados (no outliers), arriba del 60 % para tiempos ASM y C, salvo caso señalado antes, decidimos aceptar al promedio como representación de la mayoría de los datos. Se observa misma situación para 512x512 que para 16x16, con pobre ventaja para caso *2daOp*. A causa de esto hemos decidido evaluar la función usando *2daOp*, y las matrices $solver \rightarrow u$, $solver \rightarrow v$ para todos los tamaños propuestos: 16x16, 32x32, 64x64, 128x128, 256x256 y 512x512.

	C	%	SD(C)	ASM	%	SD(ASM)
<i>solver_lin_solve.1erOp_16</i>	431840.428	49	5446.690	259397.065	61	1418.848
<i>solver_lin_solve.2daOp_16</i>	311493.016	59	2738.276	288969.463	97	10882.715
<i>solver_lin_solve.3raOp_16</i>	246369.493	75	1750.180	203057.562	96	9578.348
<i>solver_lin_solve.4taOp_16</i>	211043.814	97	9477.334	149839.649	97	3777.415
<i>solver_lin_solve.1erOp_512</i>	2.345*e+08	83	4.538*e+06	1.627*e+08	76	2.517*e+06
<i>solver_lin_solve.2daOp_512</i>	2.546*e+08	70	2.528*e+06	2.153*e+08	89	2.895*e+06
<i>solver_lin_solve.3raOp_512</i>	2.401*e+08	62	2.779*e+06	1.571*e+08	75	5.485*e+05
<i>solver_lin_solve.4taOp_512</i>	2.326*e+08	80	2.384*e+06	1.586*e+08	86	1.889*e+06

Cuadro 2: Tabla de promedios ticks función Solver Lin Solve para tamaños 16x16 y 512x512. Valores con exponente fueron redondeados para mostrarse en tabla.

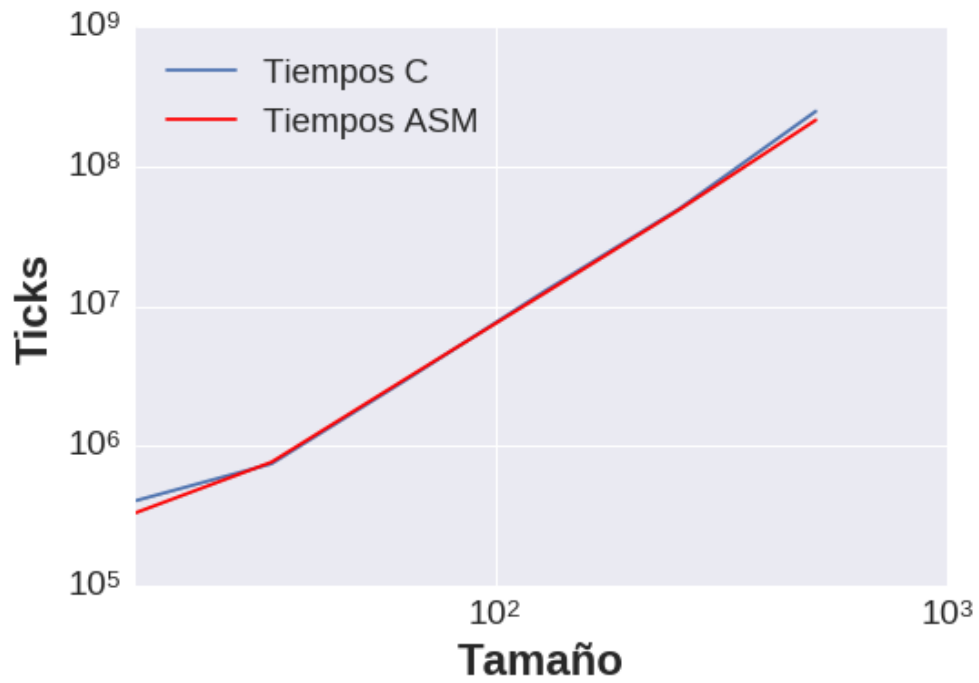
Con estos parámetros variamos el tamaño de las matrices y graficamos los tiempos (ver Figura 3(a)).

Se observa comportamiento similar de gasto temporal, con código C apenas gastando más ticks en tamaños arriba de 128x128 que ASM. Suponemos que este comportamiento parejo es a causa de que si bien se usa instrucciones SIMD en código ASM no se aprovecha del todo el proceso de datos en paralelo a causa de restricciones de código C de la función, que es la fuente de implementación ASM.

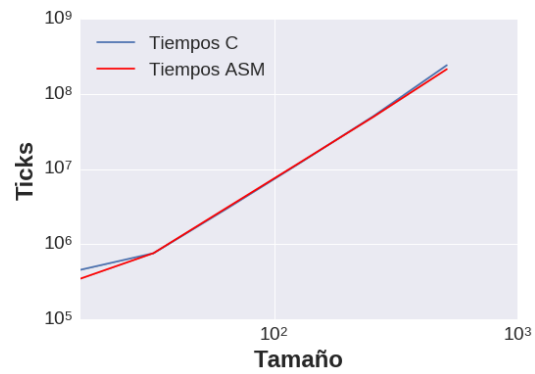
3.2.2. Compilación hecha con gcc en opción o1

Para comparar gráficas hemos decidido repetir las mediciones con los mismos parámetros que usamos en anterior medición (a , b y c de *2daOp*). El resultado se ve en Figura 3(b).

No se ve una gran diferencia en los tiempos C y ASM manteniéndose el comportamiento similar de gasto temporal al crecer en tamaño las matrices que usa la función.



(a) Compilación hecha con gcc en opción o0



(b) Compilación hecha con gcc en opción o1

(c) Compilación hecha con gcc en opción o3

Figura 3: Tiempos en ticks de ejecución de código C vs código ASM para función Solver Lin Solve.

3.2.3. Compilación hecha con gcc en opción o3

Repetimos parámetros de la función y graficamos los tiempos para distintos tamaños (Figura 3(c)).

Se ve que no hay cambios respecto a gráfica hecha con gcc en opción o0 y o1.

3.3. Función Solver Project

3.3.1. Compilación hecha con gcc en opción o0

Hemos medido la función variando parámetros 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 a posición actual, *2daOp* con mismo proceso pero comenzando en (0.2, -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 tamaño 16x16 implementación C tiende a gastar alrededor de 50 % más de clocks que código ASM. El desvío estandar nos informa que los datos se mantienen cercanos a la media, y esto, junto a el porcentaje de datos no outliers obtenido (columna % en tabla), arriba del 60 %, nos da confianza de tomar al promedio como representante de datos. En caso 512x512 se observa que C también tiende a gastar arriba de 60 % más del total que gasta ASM. El porcentaje de datos y el desvío estandar muestran comportamiento similar a caso 16x16.

	C	%	SD(C)	ASM	%	SD(ASM)
<i>solver_project_1erOp_16</i>	482456.163	98	52047.911	258971.321	56	8276.058
<i>solver_project_2daOp_16</i>	336402.153	98	19380.540	193437.0422	71	1054.726
<i>solver_project_3eraOp_16</i>	276677.958	96	9241.531	167878.729	96	4952.023
<i>solver_project_4taOp_16</i>	219557.092	97	15180.296	138860.719	89	2197.845
<i>solver_project_1erOp_512</i>	2.678*e+08	97	5.046*e+06	1.607*e+08	88	2.629*e+06
<i>solver_project_2daOp_512</i>	2.706*e+08	88	3.492*e+06	1.657*e+08	73	2.377*e+06
<i>solver_project_3raOp_512</i>	2.696*e+08	88	2.406*e+06	1.665*e+08	84	1.915*e+06
<i>solver_project_4taOp_512</i>	2.697*e+08	93	2.430*e+06	1.649*e+08	91	2.743*e+06

Cuadro 3: Tabla de promedios de ticks para función Solver Project para tamaños 16x16 y 512x512. SD es desvío estandar y % es porcentaje de datos no outliers promediados. Valores con exponente fueron redondeados para mostrarse en tabla.

A causa de este análisis hemos decidido usar matrices de caso *1erOp* para evaluar la función. En la gráfica se muestran los promedios para distintos tamaños: 16x16, 32x32, 64x64, 128x128, 256x256 y 512x512 (Figura 4(a)).

Se observa leve ventaja de tiempos de código ASM sobre código C, creciendo ambos tiempos paralelamente. Suponemos que la leve ventaja que saca ejecución de un código a otro se debe a la llamada que hace Solver Project a la función Solver Lin Solve, que presentó ínfima ventaja de la implementación ASM sobre la de C.

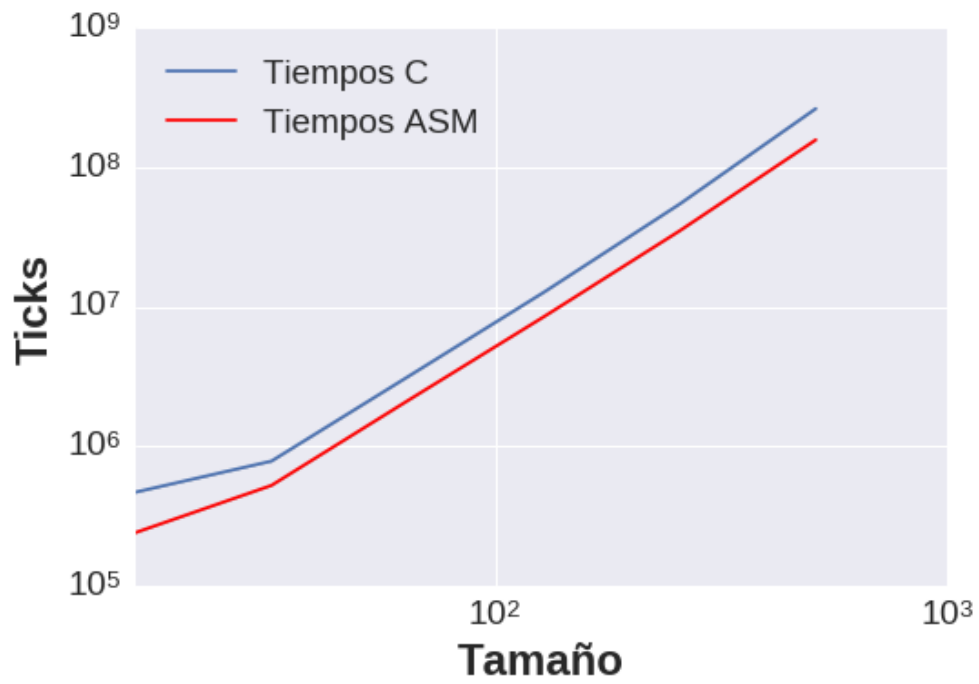
3.3.2. Compilación hecha con gcc en opción o1

Se usa misma selección de parámetros que en anterior medición, matrices de caso *1erOp*. En la Figura 4(b) se ve el resultado.

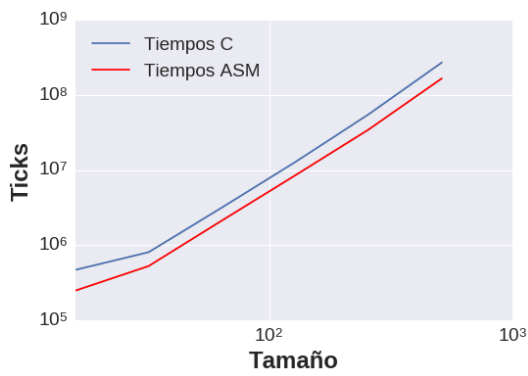
Nuevamente no se nota cambios entre tiempos compilados con gcc opción o0 y gcc opción o1.

3.3.3. Compilación hecha con gcc en opción o3

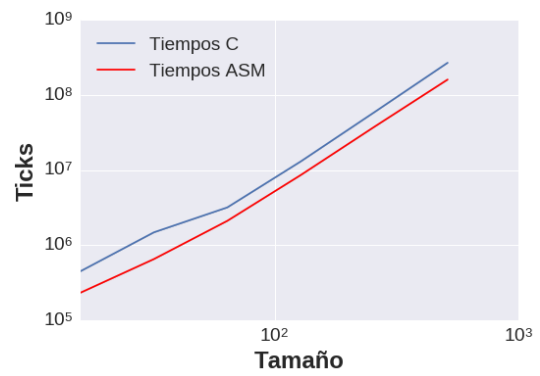
Repetimos parámetros en medición y obtuvimos el resultado de Figura 4(c).



(a) Compilación hecha con gcc en opción o0



(b) Compilación hecha con gcc en opción o1



(c) Compilación hecha con gcc en opción o3

Figura 4: Tiempos en ticks de ejecución de código C vs código ASM para función Solver Project.

No se nota mejora en tiempos de implementación C respecto a anteriores gráficas, la de Solver Project y gcc en o1 y la de Solver Project y gcc en o0.

4. Conclusión

La primera problemática encontrada fue identificar en qué casos se puede paralelizar el procesamiento y en cuáles no. Notamos que solo es posible utilizar instrucciones vectoriales cuando los datos están ubicados en forma contigua en memoria. Dada una matriz de datos cuyas filas se encuentran almacenadas en forma secuencial y contigua en memoria, se puede paralelizar el proceso si recorremos la matriz en forma horizontal (i.e. fila a fila) pero no en forma vertical (i.e. columna a columna).

En la función Solver Lin Solve la paralelización de los cálculos implicó realizar las operaciones matemáticas en distinto orden y esto trajo aparejada una diferencia en los resultados de ambas implementaciones, debida al redondeo propio de la representación numérica de punto flotante. Consideremos cuatro matrices del mismo tamaño: dos matrices p_{ASM} y p_C inicializadas con los mismos valores y otras dos matrices div_{ASM} y div_C también inicializadas con los mismos valores. Luego apliquemos la implementación en C de Solver Lin Solve a las matrices con subíndice C y la implementación en ensamblador a las matrices con subíndice ASM. El módulo de la diferencia de cada uno de los valores finales de las matrices p y div fue siempre menor que 10^{-4} en las pruebas realizadas. La conversión de los valores numéricos expresados en punto flotante de precisión simple a precisión doble antes de realizar las operaciones matemáticas no aportó una reducción visible de esta diferencia.

Notamos que la utilización de registros vectoriales permite reducir drásticamente la cantidad de accesos a memoria necesarios, lo cual, como vimos en la introducción, tiene un impacto serio en la performance de los programas. Por ejemplo, en la implementación en lenguaje ensamblador de Solver Lin Solve logramos reducir los accesos a memoria a cinco por cada cuatro celdas procesadas. Suponemos que la implementación en lenguaje C hace más accesos a memoria mirando la programación.

En un análisis de los gráficos de resultados, se puede observar que, a pesar de la variación del tamaño de las entradas, la proporción de ciclos de reloj se mantiene constante entre las implementaciones de C y ensamblador, siendo siempre menor la medición de las implementaciones en ensamblador.

En la función Solver Set Bnd se observó buena diferencia en gastos temporales, la implementación C gastó alrededor de seis veces el tiempo gastado por la implementación en ASM. En este caso se utilizan instrucciones SIMD y no hay llamadas a funciones externas.

En la función Solver Lin Solve se observó ínfima diferencia en tiempo de ejecución. Código C presentó un incremento en tiempo de ejecución de alrededor del 10 % gastado por código ASM en el peor de los casos estudiados. Suponemos que la llamada a otra función influencia en la disminución de diferencia de tiempos pero sobre todo esto se debe a la restricción que hace el código fuente, C, a la implementación ASM, es decir la imposibilidad de procesar múltiples datos en paralelo aunque por lo menos logramos aprovechar las funciones SIMD al cargar múltiples datos de memoria en los registros XMM.

En la función Solver Project se observó leve diferencia en tiempos de ejecución, implementación C gastó alrededor de un 50 % más de tiempo que código ASM. En este caso no se restringe el proceso múltiple de datos en la función y aprovechamos las funciones SIMD.

Por otra parte las optimizaciones de compilador, opción o1 y o3, no demostraron cambios notables en los resultados respecto a compilar con opción o0.