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

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 asi 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 tamaõ 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 asi realizar menos iteraciones.

2.1. solver_lin_solve

La manera en la que opera una iteracion del ciclo mas 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 l primera y avanzando de a cuatro columnas.
- Al comienzo de for i, rax quiero que apunte a la posicion (i,1) de la matriz x y r8 apunta a la misma posicion pero en la matriz x0. Tambien 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 posicion (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 pos (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 estaria el bloque que comienza en (i,j-1) de x que lo tengo de la iteracion anterior, si es que no estoy en la primera iteracion, 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. Tambien 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. Tambien 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 iteracion. Por ultimo, 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 despues sumarlo en la posicion correspondiente. Entonces lo obtenido en xmm6 lo guardo en lo apuntado por rax y en xmm4 para usarlo en la siguiente iteracion.
- Antes de terminar una iteracion de j, actualizo rax y r8 moviendolos N+2 floats. La idea es seguir en la siguiente fila.
- Antes de terminar una iteracion de i, actualizo rcx para que valga N+2, asi puedo armar rax en la siguiente iteracion. La idea es seguir con las siguientes 4 columnas.
- Hago un llamado a solver_set_bnd, pasandole los parametros apropiados. Despues del llamado reconstruyo xmm0 para que tenga cuatro float 'a' y xmm1 posea cuatros float 'c'.
- Si bien no corresponde a una iteracion, aclaro que uso los 5 registros que preservan su valor segun la convencion guardando 5 de los 6 parametros. El parametro faltante b y la variable local k las apilo en el stack(apilo primero b y despues 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 el 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:

- Primero, se van a hacer un total de (ancho * alto / 4) de iteraciones, dividimos a N por 4 porque vamos a agarrar de a 4 posiciones de la matriz.
- Vamos a diferenciar 3 casos, uno cuando estamos a principio de una fila, otro en el medio de la fila y otro al final de una fila.
- Estos casos nos van a servir a la hora de ver las posiciones $j + 1$ y $j - 1$.
- Comenzamos por traer 4 celdas de source al comienzo del ciclo.
- Si estamos al comienzo de la fila con el caso de inicio de fila, nos vamos a traer las 4 posiciones que estoy mirando, y las 4 siguientes.
- La primera posicion no nos va a interesar, porque luego la vamos a pisar, cuando llamemos al metodo que setea los bordes de la matriz.

- 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.
- Luego si estamos a mediados de la fila, vamos a tener las 4 celdas que estoy mirando, las 4 siguientes y las 4 anteriores.
- (Para poder aprovechar que las anteriores ya las vamos a tener de la iteracion anterior, las vamos almacenando en un XMM, donde guardamos las 4 celdas anteriores, asi no las tenemos que pedir a memoria nuevamente)
- Con estos 3 sets de 4 celdas, vamos a realizar SHIFTS nuevamente, para poder realizar la operacion entre las posiciones $j+1$ y $j-1$, para las cuatro celdas a la vez.
- Si entramos al caso que estamos al final de la fila, es analogo al caso de principio de fila, con la diferencia que aca vamos a traernos las 4 celdas que estamos mirando y las 4 anteriores. Operamos de la misma forma que al comienzo de la fila.
- Luego en cualquiera de los 3 casos, a la hora de operar con las posiciones $i+1$ y $i-1$, vamos a traernos en cada iteracion las 4 celdas actuales, las 4 celdas que se encuentran en la fila anterior, arriba de las actuales y las 4 celdas de la fila siguiente, que se encuentran debajo de las actuales.
- teniendo *anteriores* y *siguientes*, podemos realizar una resta entre esos dos registros, para poder computar para las 4 celdas actuales, la operacion p[IX(i+1,j)]-p[IX(i-1,j) que queriamos realizar.
- Por ultimo, dependiendo del caso, vamos a utilizar una mascara que contiene valores 0,5 y el valor N para poder completar las operaciones con los valores que ya obtuvimos para las 4 celdas que estamos mirando.
- Una vez que terminamos con el primer ciclo, vamos a llamar a *solver_set_bnd* y a *solver_lin_solve*.
- Luego vamos a construir el segundo ciclo, de la misma manera que construimos el primero, con la unica diferencia que vamos a realizar las operaciones correspondientes con las celdas que estamos mirando.
- Por ultimo llamamos al metodo *solver_set_bnd*.

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

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.

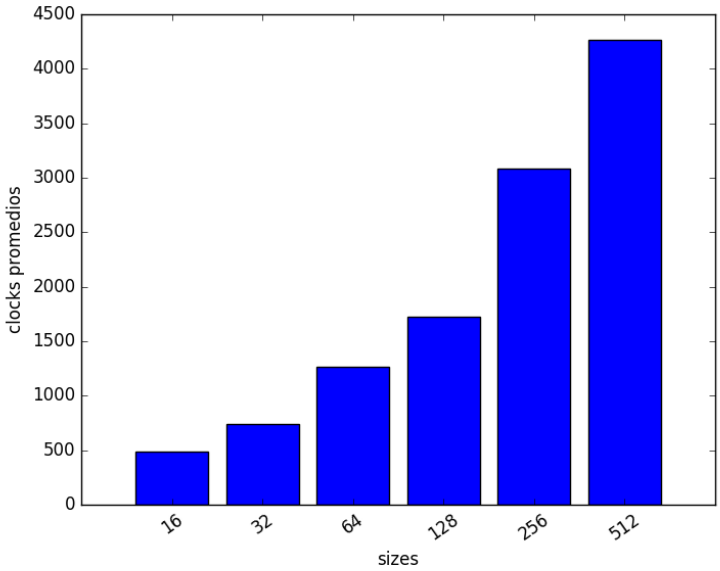

Figura 1: Código ASM con matriz U

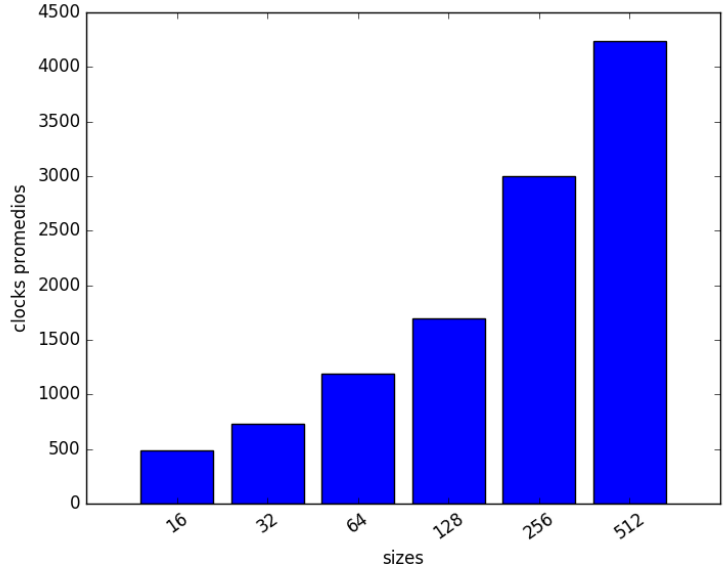

Figura 2: Código ASM con matriz V

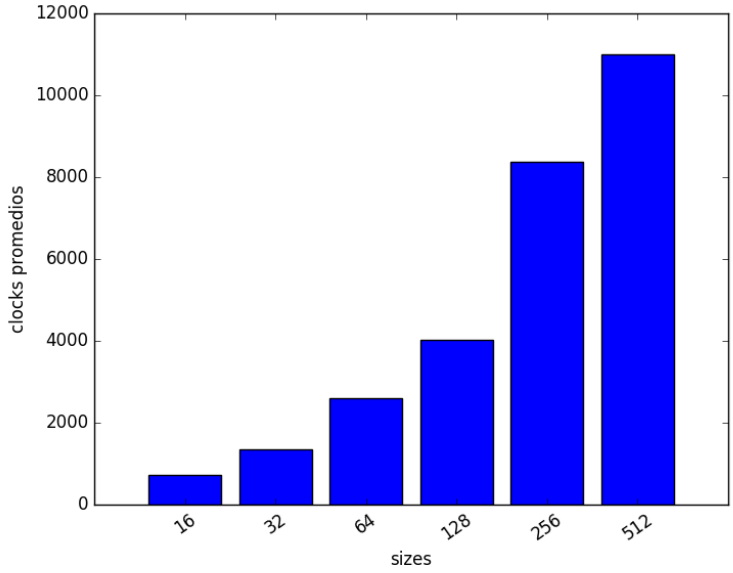

Figura 3: Código C con matriz U

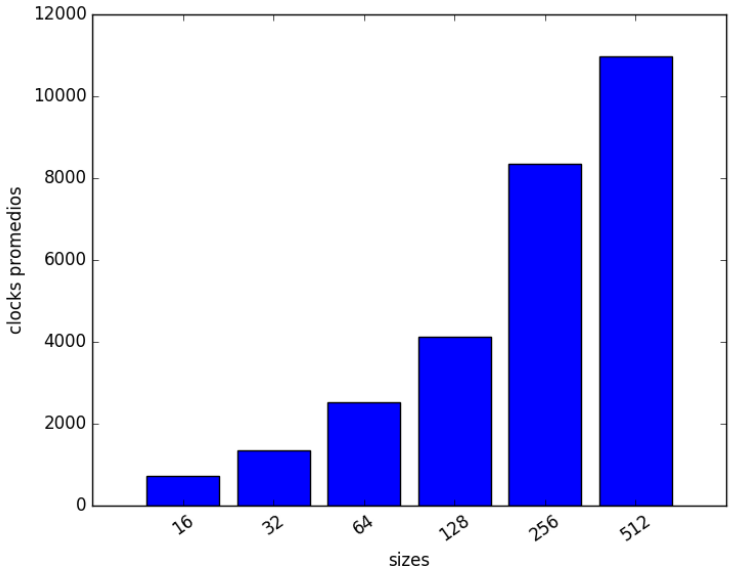

Figura 4: Código C con matriz V

3.3. solver_project

El solver

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.

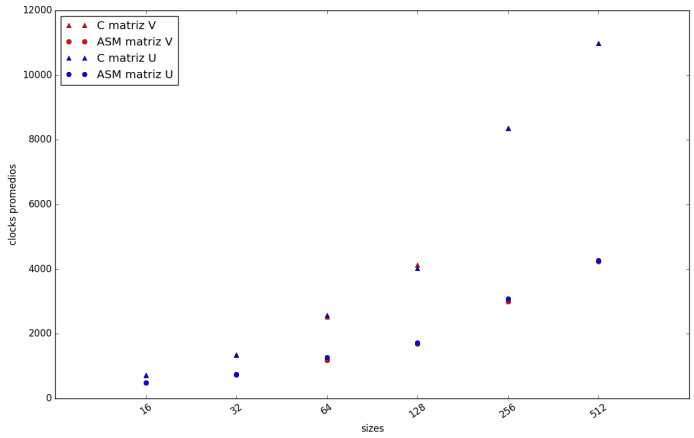

Figura 5: ASM VS C solver_set_bnd

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.

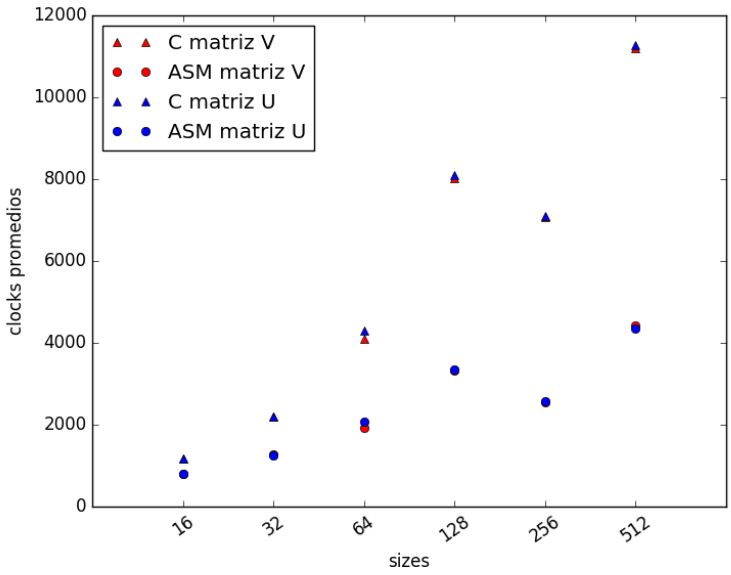

Figura 6: ASM VS C solver_set_bnd con 200 iteraciones