DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Modelo Navier Stokes 2D

20 de febrero de 2018 — Organización del Computador II

Integrante	LU	Correo electrónico
Ventura, Martín Alejandro	249/11	venturamartin90@gmail.com
Muiño, María Laura	399/11	mmuino@dc.uba.com

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

1. INTRODUCCIÓN

Los flujos son gobernados por ecuaciones diferenciales parciales, que representan las leyes de conservación de masa, momento y energía. La dinámica de fluidos computacional se encarga de resolver esas ecuaciones diferenciales utilizando técnicas de análisis numérico. Las computadoras son utilizadas para realizar los cálculos requeridos para simular la interacción entre líquidos, gases y superficies definidas por las condiciones de borde. Disponer de mas poder computacional es útil para disminuir el tiempo requerido para realizar las simulaciones, o aumentar la calidad de los resultados.

Para poder aumentar el poder computacional disponible, se utilizan a menudo, técnicas de computación en paralelo, o de computación vectorial. En este trabajo nos centraremos en la tecnología de computación vectorial SIMD (Single instruction multiple data) de Intel.

Concretamente se desarrollará codigo en assembler que utilizando las instrucciónes de vectorización de los procesadores Intel logre un aumento de rendimiento. Luego se comparará ese aumento de rendimiento con tecnicas automaticas de vectorización o paralelización, tales como OpenMP, una api para el procesamiento multinucleo con memoria compartida, y las optimizaciónes disponibles en los compiladores ICC (Intel C compiler) y g++, que a su vez utilizan instrucciónes SIMD.

Hay diversos problemas de flujo conocidos que son utilizados frecuentemente para testear aplicaciónes de este estilo. En este trabajo utilizaremos cavity flow y channel flow. describirlos

Se realizó una solución iterativa de punto fijo, creo. Por que? No hay iteracion de punto fijo en el codigo.

2. DESARROLO

2.1. Discretización. Comenzaremos con algunas definiciones. al modelar con diferencias finitas, se utilizan ciertos reemplazos de los operadores diferenciales conocidos como discretizaciones. Como su nombre indica, estos son versiones discretas de los operadores, y se los usa bajo el supuesto de que en el limite se comportan de forma similar. Pasaremos ahora a definir algunas discretizaciones que serán utilizadas en al hacer el pasaje.

Centradas de primer orden:

$$\frac{dU}{dx} = \frac{U^n_{i+1,j} - U^n_{i-1,j}}{2dx}$$

$$\frac{dU}{dy} = \frac{U^n_{i,j+1} - U^n_{i,j-1}}{2dy}$$

$$\frac{dU}{dt} = \frac{U^{n+1}_{i,j} - U^{n-1}_{i,j}}{2dt}$$

Centradas de segundo orden:

$$\frac{d^2U}{dx^2} = \frac{U^n_{i+1,j} - 2*U^n_{i,j} + U^n_{i-1,j}}{dx^2}$$

$$\frac{d^2U}{dy^2} = \frac{U^n_{i,j+1} - 2*U^n_{i,j} + U^n_{i,j-1}}{dx^2}$$

$$\frac{d^2U}{dt^2} = \frac{U^{n+1}_{i,j} - 2*U^n_{i,j} + U^{n-1}_{i,j}}{dt^2}$$

Adelantadas de primer orden:

$$\frac{dU}{dx} = \frac{U^n_{i+1,j} - U^n_{i,j}}{dx}$$

$$\frac{dU}{dy} = \frac{U^n_{i,j+1} - U^n_{i,j}}{dy}$$

$$\frac{dU}{dt} = \frac{U^{n+1}_{i,j} - U^n_{i,j}}{dx}$$

Atrasadas de primer orden:

$$\frac{dU}{dx} = \frac{U^n_{i,j} - U^n_{i-1,j}}{dx}$$

$$\frac{dU}{dy} = \frac{U^n_{i,j} - U^n_{i,j-1}}{dy}$$

$$\frac{dU}{dt} = \frac{U^n_{i,j} - U^{n-1}_{i,j}}{dx}$$

Reemplazando estas discretizaciones en las ecuaciones semi-acopladas de Navier Stokes y obtenemos:

$$\frac{u_{i,j}^{n+1}-u_{i,j}^{n}}{\Delta t}+u_{i,j}^{n}\frac{u_{i,j}^{n}-u_{i-1,j}^{n}}{\Delta x}+v_{i,j}^{n}\frac{u_{i,j}^{n}-u_{i,j-1}^{n}}{\Delta y}=$$
$$-\frac{1}{\rho}\frac{p_{i+1,j}^{n}-p_{i-1,j}^{n}}{2\Delta x}+\nu\left(\frac{u_{i+1,j}^{n}-2u_{i,j}^{n}+u_{i-1,j}^{n}}{\Delta x^{2}}+\frac{u_{i,j+1}^{n}-2u_{i,j}^{n}+u_{i,j-1}^{n}}{\Delta y^{2}}\right)+Fu$$

$$\frac{v_{i,j}^{n+1}-v_{i,j}^{n}}{\Delta t}+u_{i,j}^{n}\frac{v_{i,j}^{n}-v_{i-1,j}^{n}}{\Delta x}+v_{i,j}^{n}\frac{v_{i,j}^{n}-v_{i,j-1}^{n}}{\Delta y}=$$
$$-\frac{1}{\rho}\frac{p_{i,j+1}^{n}-p_{i,j-1}^{n}}{2\Delta y}+\nu\left(\frac{v_{i+1,j}^{n}-2v_{i,j}^{n}+v_{i-1,j}^{n}}{\Delta x^{2}}+\frac{v_{i,j+1}^{n}-2v_{i,j}^{n}+v_{i,j-1}^{n}}{\Delta y^{2}}\right)+Fv$$

$$\frac{p_{i+1,j}^{n}-2p_{i,j}^{n}+p_{i-1,j}^{n}}{\Delta x^{2}}+\frac{p_{i,j+1}^{n}-2*p_{i,j}^{n}+p_{i,j-1}^{n}}{\Delta y^{2}}=\rho\left[\frac{1}{\Delta t}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}+\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\right)\right.$$

Aquí en la ultima ecuación podemos ver que no se reemplazó directamente cada operador mediante las ecuaciones de discretización, sino que se agregó un termino temporal, sin que hubiera en principio información sobre el tiempo en la ecuación de la presión. Este cambio se hace con el objetivo de terminar de acoplar la ecuación de la presión con las ecuaciones de velocidad. La derivación de esta solución no se presentará en este trabajo.

Cabe aclarar que al discretizar, se puede modelar el sistema mediante un método implícito o explicito. Un método implícito, o parcialmente implícito, incluiría una ponderación entre los valores de las variables en la iteración n, y la iteración n+1. En este trabajo utilizaremos un método explicito, ya que el sistema de ecuaciones determinado por un método explicito es lineal, y resulta en relaciones donde un elemento en la iteración n+1 depende de otros en la iteración n, pudiendo entonces realizarse los reemplazos en las matrices que representan el sistema de forma directa, y resultando así en una implementación mas sencilla. Un método implícito da como resultado un sistema no lineal, en el cual hay que hacer uso de algún método de resolución de sistemas no lineales, como punto fijo, lo cual aumenta la complejidad de la implementación.

2.2. Implementación. La implementación fue realizada casi completamente en C++, excepto por la sección donde es critico el rendimiento, la cual fue programada en C++ y Assembler. Esta sección es la correspondiente a la función calcVelocities, que como su nombre indica, calcula las velocidades en cada punto.

El programa define las matrices U2, U2, V1, V2, P1, P2, que representan el estado del sistema en una iteración para la velocidad en u, en v, y la presión, y luego estas mismas en la iteración siguiente.

Se definen las condiciones iniciales del problema, y luego se utiliza un método explicito para calcular los nuevos valores del sistema. Estos son guardados en U2, V2, y P2. Seguido

de esto el programa reemplaza los valores de U1, V1, y P1, por aquellos de U2, V2 y P2, quedado así preparado para la siguiente iteración.

Se implementó también una clase mat2, que representa una matriz, y que contiene un puntero a un arreglo de números de punto flotante de simple precisión y dos enteros que representan el tamaño en filas y columnas de la matriz. Ademas la clase cuenta con funciones que realizan la abstracción de indexar en el arreglo calculando la posición del elemento buscado como la columna pedida, más la fila pedida multiplicada por la cantidad de columnas.

En cuanto a la vectorización, como se comentó anteriormente se utilizó la tecnología SIMD de Intel, de la forma descripta a continuación:

- Mediante una directiva DEFINE presente en el Makefile, se elije si se desea compilar con soporte para SIMD, soporte para OpenMP, ambos, o ninguno.
- El programa define las matrices necesarias con los valores iniciales segun lo estipulado por el metodo de discretización utilizado.
- La sección del programa que realiza el calculo consta de tres ciclos for consecutivos. El primero cicla en la variable t, que representa el tiempo. el segundo en la variable i, que representa la altura, y el tercero en la variable j que representa el ancho.
- Mediante la utilización de las directivas de compilador, el codigo compilado constara de una implementación en C++ plano, una implementación SIMD, donde al llegar a un valor menor al ancho de los registros XMM dividido por el tamaño de el tipo de datos flotante de precision simple se cambia el procesamiento mediante SIMD por el de C++, y ademas mediante estas mismas directivas puede definirse o no la presencia de OpenMP, logrando asi la utilización de multiples nucleos.
- La paralelización mediante OpenMP se realiza en la variable i.
- La vectorización mediante SIMD, se realiza en la variable j. Es decir, en un solo llamado a la versión de assembler de la funcion de calculo se calculan 4 elementos consecutivos en memoria.
- Ademas, durante la simulación no se crean ni se destruyen matrices, sino que estas son reutilizadas cambiando los valores que contienen para no perder tiempo manejando memoria.

3. EXPERIMENTACION

3.1. Análisis del código generado. Usando la herramienta objdump sobre los archivos objeto (.o) del código de c++ (sin flags de optimización), obtuvimos y analizamos el código ensamblado por el compilador. Notamos las siguientes caracterísicas del código generado que dan lugar a mejoras en el rendimiento:

- Dentro de la función calcVelocities, la función donde se ralizan los cálculos que luego se vectorizarán, hay llamados a líneas consecutivas.
- Hay consultas a memorias innecesarias, por ejemplo, se pide un mismo valor a memoria varias veces, a pesar de haber sido guardado en un registro y nunca haber sido reemplazado con otro valor.
- Se manejan las variables locales almacenandolas en la pila, mientras que sólo se usan los registros de manera auxiliar para realizar operaciones.

3.2. Optimizaciones del compilador. El compilador de gcc posee una gran cantidad de optimizaciones. Un grupo de estas optimizaciones es habilitado por el parámetro -O1. Entre ellos se encuentran los siguientes flags:

- fdce: Realiza eliminación de código muerto en RTL [1]. Disminuyendo asi la presencia de codigo que no tiene efecto en el resultado.
- fdse: Realiza eliminación de guardado muerto en RTL, valores que son escritos a memoria, pero que no vuelven a leerse.
- fsplit-wide-types: Cuando se usa un tipo de datos que ocupa múltiples registros, como por ejemplo l̈ong longën un sistema de 32-bit, separa el dato y lo guarda de manera independiente. Esto, normalmente, genera mejor código para estos tipos de datos, pero hace más dificil el debuggeo.
- -fmerge-constants: Intenta unir constantes identicas (cadenas o flotantes) a travez de unidades de compilación. Esta opción es la por defecto para la compilacion optimizada si el ensamblador y linker la soportan.

SEGUIR VIENDO EL RESTO DE LOS FLAGS, QUIZA HAYA ALGUNOS QUE JUSTIFIQUEN LOS RES

Compilamos el código de C++ con el flag de optimización -O1 y se obtuvieron los siguientes resultados respecto al código sin flags de optimización:

	Tiempo ejecución	
Tamaño	Sin optimización	Con optimización (O1)
6 x 6	8.328s	3.260s
8 x 8	14.640s	5.776s
10 x 10	22.872s	8.980s
12 x 12	32.916s	12.888s

Los tiempos de ejecución se reducen utilizando el flag de optimización.

[1] Register Transfer Language. Es una representación intermedia (RI), similar a assembler. Se utiliza para describir el transferencia de datos de una arquitectura a nivel registro

Veamos que, al contrario que el tiempo de ejecución, el tiempo de compilación del código aumenta.

Tiempo compilación	
Sin optimización	Con optimización (O1)
0.652s	1.044s

Utilizamos nuevamente la herramienta objdump. El código en asembler obtenido mediante la herramienta era reducido en cuanto a cantidad de líneas (por ejemplo, la función calcVelocities tenia 558 líneas en assembler sin flags de optimización y luego con el flag 341 líneas).

Además notamos que, con el uso del parámetro -O1, se hace uso de los registros para el manejo de las variables locales en vez de la pila como sucedía usando el parámetro -O0 (está por defecto).

Por otro lado, está el análisis del uso de memoria. TODO: Corri valgrind pero todavia no terminé de analizar los resultados, es medio dolor de cabeza.

3.3. Comparación entre secuencial, vectorial y multicore.

3.4. CPU vs. memoria. El objetivo de este experimento es conocer el factor que limita la performance en el código de asembler; la intensidad del cómputo o la cantidad de accesos a memoria.
El experimento sugiere agregar ciclos bobos en el código de asembler y ver si el tiempo de uso de la cpu tiene picos.

Si los picos de tiempo se hacen ver, quiere decir que el procesamiento es nuestro cuello de botella.

En caso contrario hay que analizar el uso de la memoria. En este caso, yo creo que lo más costoso es el acceso a memoria, pero no lo verifique todavía.

Realicé una medicion con valgrind-massif, que mide el uso del heap de memoria y tira varios detalles.

El pico más grande sucede en el tiempo 109995366147 (?) donde el uso total es de 6,380,856B - el uso posta es de 6,380,676B y el uso auxiliar es de 180B.

El pico se realiza cuando se llama a mat2::inic(float)

TODO: AVERIGUAR QUE ESTA PASANDO CON ESTE LLAMADO RARO. PINTA QUE ES SOLO CUANDO SE CREA LAS MATRICES, EL PROBLEMA ES QUE NO SUCEDE EN CADA ITERACION O AL PPIO DEL PROGRAMA, QUE ES DONDE SE ESPERARIA QUE SE CREEN TODAS LAS MATRICES, SINO AL FINAL DE LOS SNAPSHOTS. QUIZA INTERPRETE MAL LA SALIDA DE MASSIF, YA QUE EL HISTOGRAMA MUESTRA UN PICO SUPER GRANDE AL PPIO

4. Conclusión