DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Modelo Navier Stokes 2D

25 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.

3.2.1. Optimizaciones O1. 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.
- -fdelayed-branch: No tiene efecto en el código pero causa la ejecución a priori en ramas de ejecución para aumentar la performance.

Tenemos un extracto de código de la función CalcVelocities (donde se realiza el grueso de los cálculos) donde podemos ver el flag de eliminación de código muerto (fdce) en acción. Tenemos un primer pedazo de código donde obtenemos la ejecucion de cpp sin optimización en version assembler y más abajo lo mismo pero con optimización -O1. El formato es línea de código, instrucción en hexadecimal y descripción de la instrucción versión asembler.

Muestra de código de cpp

```
<_simulatorCalcVelocities>:
    1abe: 55                    push   rbp
    1abf: 48 89 e5              mov    rbp,rsp
    1ac2: 48 83 ec 38           sub    rsp,0x38
    1ac6: 48 89 7d e8           mov    QWORD PTR [rbp-0x18],rdi
```

[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

```
1aca: 89 75 e4              mov    DWORD PTR [rbp-0x1c],esi
1acd: 89 55 e0              mov    DWORD PTR [rbp-0x20],edx
1ad0: 48 8b 45 e8           mov    rax,QWORD PTR [rbp-0x18]
1ad4: 48 8d 88 e0 00 00 00  lea    rcx,[rax+0xe0]
1adb: 8b 55 e0              mov    edx,DWORD PTR [rbp-0x20]
1ade: 8b 45 e4              mov    eax,DWORD PTR [rbp-0x1c]
1ae1: 89 c6                 mov    esi,eax
1ae3: 48 89 cf              mov    rdi,rcx
1ae6: e8 00 00 00 00        call   1aeb
1aeb: f3 0f 11 45 d8        movss  DWORD PTR [rbp-0x28],xmm0
1af0: 48 8b 45 e8           mov    rax,QWORD PTR [rbp-0x18]
1af4: 48 8d 88 e0 00 00 00  lea    rcx,[rax+0xe0]
1afb: 8b 55 e0              mov    edx,DWORD PTR [rbp-0x20]
1afe: 8b 45 e4              mov    eax,DWORD PTR [rbp-0x1c]
1b01: 89 c6                 mov    esi,eax
1b03: 48 89 cf              mov    rdi,rcx
1b06: e8 00 00 00 00        call   1b0b
.
.
.
2400: f3 0f 10 45 d8        movss  xmm0,DWORD PTR [rbp-0x28]
2405: 89 c6                 mov    esi,eax
2407: e8 00 00 00 00        call   240c
240c: 90                    nop
240d: c9                    leave
240e: c3                    ret
240f: 90                    nop
```

Muestra de código de cpp con 01

```
<_simulatorCalcVelocities>:
12f0: 41 57                 push   r15
12f2: 41 56                 push   r14
12f4: 41 55                 push   r13
12f6: 41 54                 push   r12
12f8: 55                    push   rbp
12f9: 53                    push   rbx
12fa: 48 83 ec 30           sub    rsp,0x30
12fe: 48 89 fb              mov    rbx,rdi
1301: 89 f5                 mov    ebp,esi
1303: 41 89 d4              mov    r12d,edx
1306: 4c 8d bf e0 00 00 00  lea    r15,[rdi+0xe0]
130d: 4c 89 ff              mov    rdi,r15
1310: e8 00 00 00 00        call   1315
1315: 0f 28 e0              movaps xmm4,xmm0
1318: f3 0f 10 7b 1c        movss  xmm7,DWORD PTR [rbx+0x1c]
131d: f3 0f 10 5b 14        movss  xmm3,DWORD PTR [rbx+0x14]
1322: f3 0f 11 7c 24 04     movss  DWORD PTR [rsp+0x4],xmm7
1328: 0f 28 c7              movaps xmm0,xmm7
132b: f3 0f 11 5c 24 10     movss  DWORD PTR [rsp+0x10],xmm3
1331: f3 0f 5e c3           divss  xmm0,xmm3
1335: 0f 28 f0              movaps xmm6,xmm0
```

```
1338: f3 0f 11 24 24        movss  DWORD PTR [rsp],xmm4
133d: f3 0f 59 f4           mulss  xmm6,xmm4
1341: f3 0f 11 74 24 08     movss  DWORD PTR [rsp+0x8],xmm6
1347: 8d 45 ff              lea    eax,[rbp-0x1]
134a: 44 89 e2              mov    edx,r12d
134d: 89 44 24 14           mov    DWORD PTR [rsp+0x14],eax
1351: 89 c6                 mov    esi,eax
1353: 4c 89 ff              mov    rdi,r15
1356: e8 00 00 00 00        call   135b
135b: f3 0f 10 24 24        movss  xmm4,DWORD PTR [rsp]


187a: 48 8d bb 30 01 00 00  lea    rdi,[rbx+0x130]
1881: 44 89 e2              mov    edx,r12d
1884: 89 ee                 mov    esi,ebp
1886: e8 00 00 00 00        call   188b
188b: 48 83 c4 30           add    rsp,0x30
188f: 5b                    pop    rbx
1890: 5d                    pop    rbp
1891: 41 5c                 pop    r12
1893: 41 5d                 pop    r13
1895: 41 5e                 pop    r14
1897: 41 5f                 pop    r15
1899: c3                    ret
```

Lo que notamos de los dos extractos de código anterior, la diferencia, es el uso de la instrucción not. Aparentemente no tiene ningun tipo de efecto, con lo cual, en el código optimizado se hace eliminación de este.

En la posición lad0 se carga el registro rax con un valor, no se lo pisa y luego vuelve a cargarlo en laf0. Este tipo de instrucciones donde no es necesario volver a cargar de memoria datos, se arregla con el flag fdse.(accesos al pedo de O0)

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).

En el código de cpp con flag O1, el manejo de memoria no tiene el comportamiento tonto de volver a cargar algo desde memoria que ya tenía. Sin embargo repite movimientos de datos innecesarios entre registros.

También notamos que no utiliza al máximo los registros, ya que guarda ciertos datos a memoria.

Además aqui se puede notar el uso de la pila para las variables en vez de usar registros en la versión no optimizada (referencia a experimento anterior). A diferencia de esto, una vez que activamos la optimización o1 podemos ver que los valores de los registros son guardados dentro de la función para permitir operaciones más rápidas por el resto del llamado.

El siguiente extracto, muestra el nivel de mejora en cuanto a manejo de datos de memoria de O1. No asi las cagadas, pero esta bueno ponerlo.

```
000000000000018e <_ZN4mat23setEiif>:
    18e: 55                 push   rbp
    18f: 48 89 e5           mov    rbp,rsp
```

```
     192: 48 89 7d f8           mov    QWORD PTR [rbp-0x8],rdi
     196: 89 75 f4              mov    DWORD PTR [rbp-0xc],esi
     199: 89 55 f0              mov    DWORD PTR [rbp-0x10],edx
     19c: f3 0f 11 45 ec        movss  DWORD PTR [rbp-0x14],xmm0
     1a1: 48 8b 45 f8           mov    rax,QWORD PTR [rbp-0x8]
     1a5: 48 8b 10              mov    rdx,QWORD PTR [rax]
     1a8: 48 8b 45 f8           mov    rax,QWORD PTR [rbp-0x8]
     1ac: 8b 40 0c              mov    eax,DWORD PTR [rax+0xc]
     1af: 0f af 45 f4           imul   eax,DWORD PTR [rbp-0xc]
     1b3: 89 c1                 mov    ecx,eax
     1b5: 8b 45 f0              mov    eax,DWORD PTR [rbp-0x10]
     1b8: 01 c8                 add    eax,ecx
     1ba: 48 98                 cdqe
     1bc: 48 c1 e0 02           shl    rax,0x2
     1c0: 48 01 d0              add    rax,rdx
     1c3: f3 0f 10 45 ec        movss  xmm0,DWORD PTR [rbp-0x14]
     1c8: f3 0f 11 00           movss  DWORD PTR [rax],xmm0
     1cc: 90                    nop
     1cd: 5d                    pop    rbp
     1ce: c3                    ret
     1cf: 90                    nop

00000000000000d4 <_ZN4mat23setEiif>:
      d4: 0f af 77 0c           imul   esi,DWORD PTR [rdi+0xc]
      d8: 01 f2                 add    edx,esi
      da: 48 63 d2              movsxd rdx,edx
      dd: 48 8b 07              mov    rax,QWORD PTR [rdi]
      e0: f3 0f 11 04 90        movss  DWORD PTR [rax+rdx*4],xmm0
      e5: c3                    ret
```

Compilamos el código de C++ con el flag de optimización -O1 y se obtuvieron los siguientes resultados en medición de tiempo de ejecución 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.

3.2.2. Optimizaciones O2.

3.3. Comparación entre secuencial, vectorial y multicore. Ya analizado el tipo de mejoras que son implementadas por G++ al utilizar -O1, se comparan mediciones de tiempo de los distintos flags presentes en el compilador, con otras formas de mejorar el rendimiento. En particular se estudia vectorización medinte SIMD y multicore mediante OpenMP.

Para cada mejora se experimentó con distintos tamaños del sistema simulado, yendo desde 1x1m^2 hasta 20x20m^2. Además, para cada tamaño, se realizaron 100 repticiónes, y se tomo la media y varianza de las mismas, con el objetivo de eliminar el error de medición introducido por la falta de control del tiempo otorgado a las distintas tareas del sistema operativo por parte del scheduler.

3.4. Tiempos de compilación y tamañano de código generado.

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).

3.5. 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