|  |  |  |
| --- | --- | --- |
| **2º curso / 2º cuatr.**  **Grado Ing. Inform.** |  | **Arquitectura de Computadores (AC)**  **Cuaderno de prácticas.**  **Bloque Práctico 5. Optimización de código**  Estudiante (nombre y apellidos): Brian Sena Simons  Grupo de prácticas y profesor de prácticas: 2ºB  Fecha de entrega: ¿¿?¿  Fecha evaluación en clase: ¿?¿? |

Antes de comenzar a realizar el trabajo de este cuaderno consultar el fichero con los normas de prácticas que se encuentra en SWAD

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo):

AuthenticAMD – AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx.

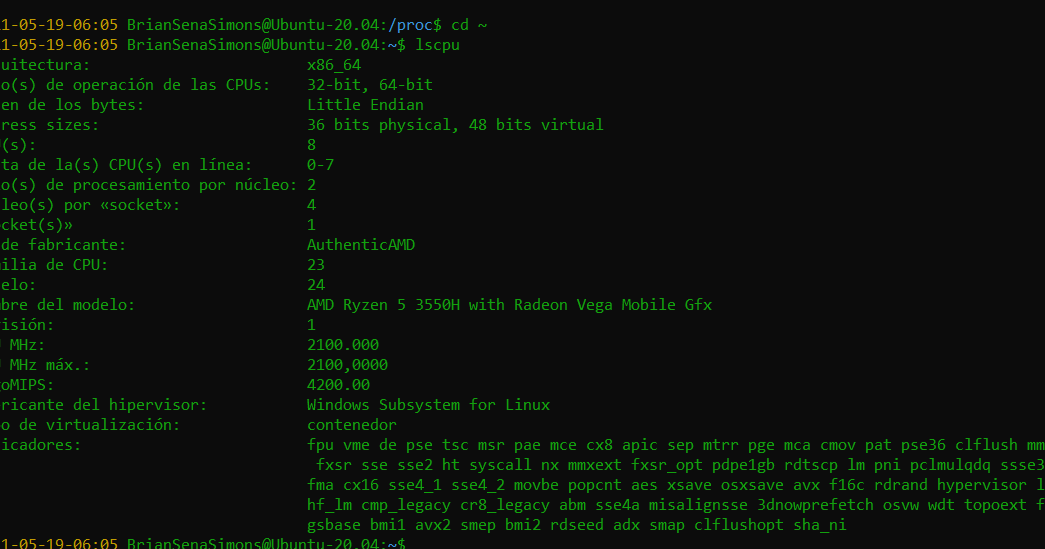
Sistema operativo utilizado:

*Windows subsystem for Linux version 1.0.*

Versión de gcc utilizada:

gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

Volcado de pantalla que muestre lo que devuelve lscpu en la máquina en la que ha tomado las medidas:

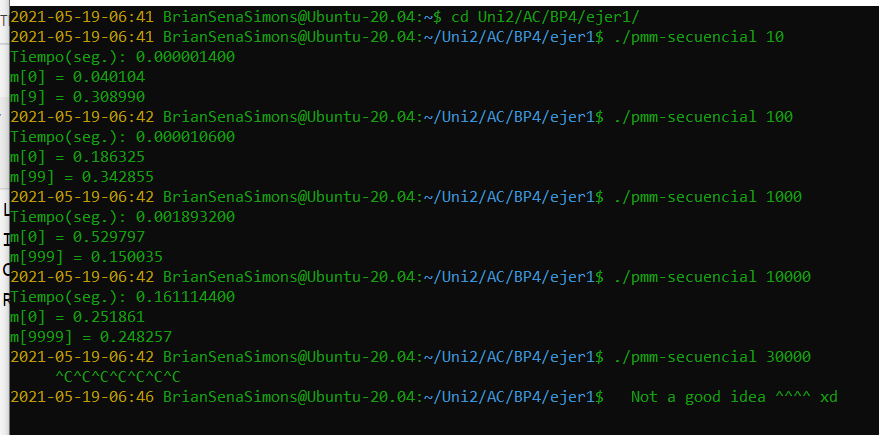
****

1. **(a)** Implementar un código secuencial que calcule la multiplicación de dos matrices cuadradas. Utilizar como base el código de suma de vectores de BP0. Los datos se deben generar de forma aleatoria para un número de filas mayor que 8, como en el ejemplo de BP0, se puede usar drand48()).

**MULTIPLICACIÓN DE MATRICES:**

**CAPTURA CÓDIGO FUENTE**: pmm-secuencial.c

|  |
| --- |
|  |



**(b)** Modificar el código (solo el trozo que calcula la multiplicación) para reducir el tiempo de ejecución. Justificar los tiempos obtenidos (usando siempre -O2) a partir de la modificación realizada. Incorporar los códigos modificados en el cuaderno.

**MODIFICACIONES REALIZADAS (al menos dos modificaciones):**

**Modificación A) –explicación-: (**Diap. 9 Seminario 4). Realizar el bucle de 4 en 4.

**Modificación B) –explicación-:** (Diap. 16 Seminario 4). Localidad de los accesos, cambiamos los índices de acceso a las matrices.

**...**

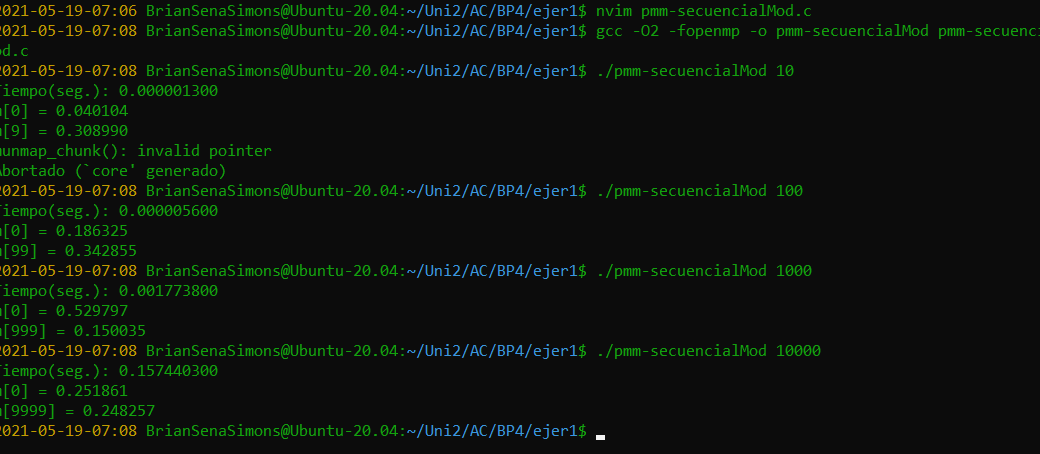
**CÓDIGOS FUENTE MODIFICACIONES**

**A) Captura de** pmm-secuencial-modificado\_A.c

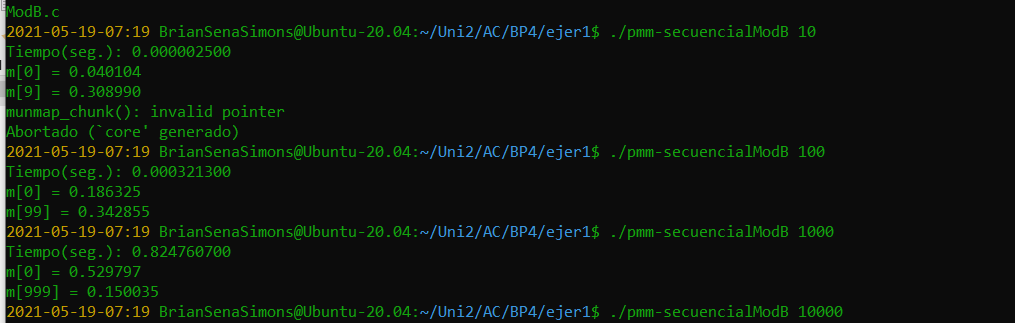
|  |
| --- |
| El Problema es cuando N no es divisible por 4.    Modificación B: |

**Capturas de pantalla (que muestren la compilación y que el resultado es correcto):**

**Modificación A:**

****

**Modificación B:**

****

**TIEMPOS:**

|  |  |  |
| --- | --- | --- |
| **Modificación** | **Breve descripción de las modificaciones** | **-O2** |
| Sin modificar |  | *De 0,25 a 10 seg. aquí* |
| Modificación A) | Desenrollado del bucle interior | 9.5x10^-4 segs menos en media. |
| Modificación B) | Cambio de Índices del bucle + Desenrollado | No Bueno, tarda más |
| ... |  |  |
|  |  |  |

**COMENTARIOS SOBRE LOS RESULTADOS Y JUSTIFICACIÓN DE LAS MEJORAS EN TIEMPO:**

Al utilizar un incremento de 4 en 4 nos ahorramos varias instrucciones a código máquina reduciendo el número de vueltas que da el bucle. Sobre todo para un N muy grande.

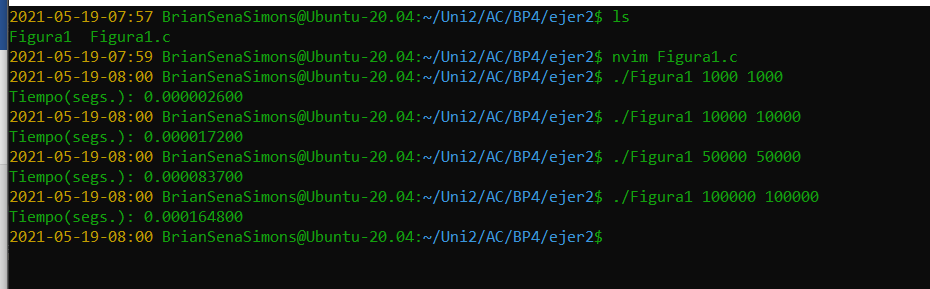
Luego el intercambio de índices era para poder jugar con el principio de localidad espacial al que se rigen las memorias. Sin embargo, con mi arquitectura no ha provocado ninguna mejora, si no que la solución de la modificación A de por si, ha sido la mejor.

1. **(a)** Usando como base el código de BP0, generar un programa para evaluar un código de la Figura 1. M y N deben ser parámetros de entrada al programa. Los datos se deben generar de forma aleatoria para valores de M y N mayores que 8, como en el ejemplo de BP0.

**CÓDIGO FIGURA 1:**

**CAPTURA CÓDIGO FUENTE**: figura1-original.c

|  |
| --- |
|  |

****

**Figura 1** .     Código C++ que suma dos vectores. M y N deben ser parámetros de entrada al programa, usar valores mayores que 1000 en la evaluación.

|  |
| --- |
| struct {          int a;          int b;  }  s[N];    main()  {     …     for (ii=0; ii<M;ii++) {        X1=0; X2=0;        for(i=0; i<N;i++)  X1+=2\*s[i].a+ii;        for(i=0; i<N;i++)  X2+=3\*s[i].b-ii;          if (X1<X2)  R[ii]=X1  else  R[ii]=X2;     }     …  } |

**(b)** Modificar el código C (solo el trozo a evaluar) para reducir el tiempo de ejecución. Justificar los tiempos obtenidos (usando siempre -O2) a partir de la modificación realizada. En las ejecuciones de evaluación usar valores de N y M mayores que 1000. Incorporar los códigos modificados en el cuaderno.

**MODIFICACIONES REALIZADAS (al menos dos modificaciones):**

**Modificación A) –explicación-:** Cambiamos los operadores \* por << Que son más eficientes a nivel máquina.

**Modificación B) –explicación-:** Cambiamos el struct por el tipo de acceso que estamos implementando. (Diap. 15 Seminario 4).

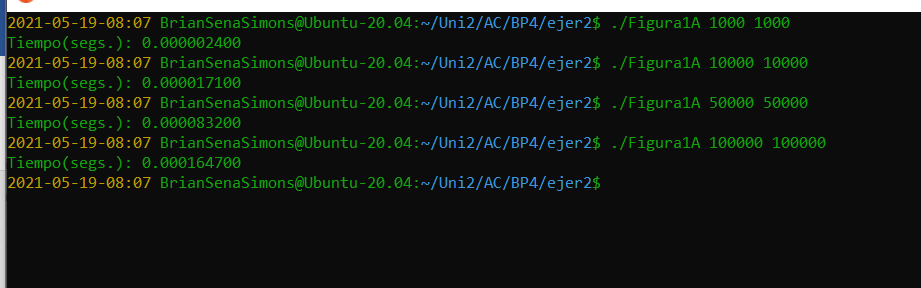
**Modificación C) -explicación:**  Cambiamos el if/else para reducir el número de saltos.

**CÓDIGOS FUENTE MODIFICACIONES**

**A) Captura** figura1-modificado\_A.c

|  |
| --- |
| Modificación B:    Modificación C: |

**Capturas de pantalla (que muestren la compilación y que el resultado es correcto):**



****

****

**TIEMPOS:**

|  |  |  |
| --- | --- | --- |
| **Modificación** | **Breve descripción de las modificaciones** | **-O2** |
| Sin modificar |  | *De 0,25 a 10 seg. aquí* |
| Modificación A) | Cambiamos los operadores \* por << Que son más eficientes a nivel máquina. | Pequeña mejora. |
| Modificación B) | Cambiamos el struct por el tipo de acceso que estamos implementando. (Diap. 15 Seminario 4). | Mismo tiempo de ejecución. |
| Modificación C) | Cambiamos el if/else para reducir el número de saltos. | Mismo Tiempo de ejecución. |
|  |  |  |

**COMENTARIOS SOBRE LOS RESULTADOS Y JUSTIFICACIÓN DE LAS MEJORAS EN TIEMPO:**

Pues al final, en mi arquitectura al menos, los cambios realizados han tenido poco impacto en el tiempo de ejecución general de la aplicación. Primeramente hemos intentado optimizar las operaciones que realizaba la máquina, y a continuación hemos probado mejorar el acceso a memoria mediante el alineamiento de datos y abusando de la localidad temporal y espacial, y por último hemos reducido la probabilidad de saltos que podría realizar el código. Sin embargo, el tiempo de ejecución seguía siendo relativamente el mismo. Una posible explicación es que mi sistema ya optimice muchas de esas cosas internamente al compilar.

1. El benchmark Linpack ha sido uno de los programas más ampliamente utilizados para evaluar las prestaciones de los computadores. De hecho, se utiliza como base en la lista de los 500 computadores más rápidos del mundo (el Top500 Report). El núcleo de este programa es una rutina que opera con flotantes de doble precisión denominada DAXPY (***D****ouble precision- real* ***A****lpha* ***X******P****lus* ***Y***) que multiplica un vector por una constante y los suma a otro vector (Lección 3/Tema 1):

for (i=0;i<N;i++) y[i]= a\*x[i] + y[i];

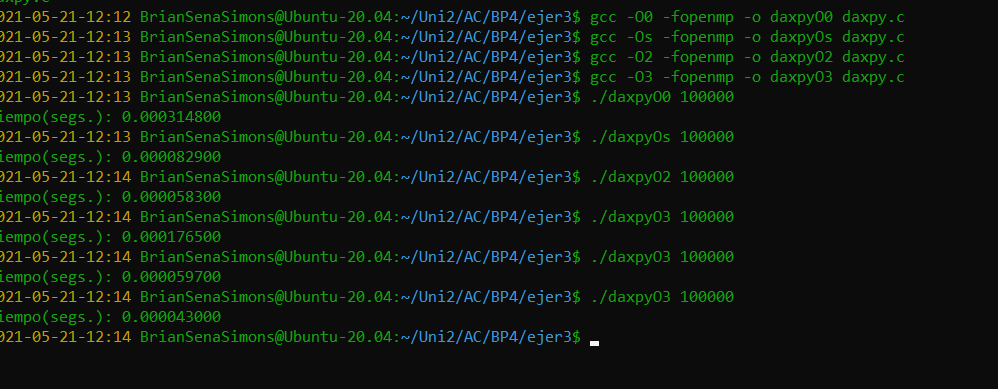
Generar los programas en ensamblador para cada una de las siguientes opciones de optimización del compilador: -O0, -Os, -O2, -O3. Explique las diferencias que se observan en el código justificando al mismo tiempo las mejoras en velocidad que acarrean. Incorporar los códigos al cuaderno de prácticas y destacar las diferencias entre ellos. Sólo se debe evaluar el tiempo del núcleo DAXPY. N deben ser parámetro de entrada al programa.

**CAPTURA CÓDIGO FUENTE**: daxpy.c

|  |
| --- |
|  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tiempos ejec.**  **Longitud vectores=100000** | **-O0** | **-Os** | **-O2** | **-O3** |
| *0.000254100* | 0.000058200 | 0.000057600 | 0.000037200 |

**(La captura no coincide con los tiempos de ejecución, ya que estos fueron calculados a continuación con varias ejecuciones para asegurar la fiabilidad de los resultados).**

****

**COMENTARIOS QUE EXPLIQUEN LAS DIFERENCIAS EN ENSAMBLADOR:**

El código generado por -O0 utiliza el puntero de pila para almacenar e ir manipulando los valores a la vez que realiza más saltos incondicionales que los demás. -Os destaca por no utilizar el puntero de pila y utiliza operaciones menos costosas para vaciar los registros como es un “xor” pero sigue fallando en el número de saltos. La opción -O2 se aprovecha de los beneficios de la -Os teniendo en cuenta ahora evitar saltos innecesarios. La -O3 presenta mayor complejidad a nivel de instrucción con intención de aumentar el rendimiento total (Éxito) cumpliendo con lo necesario para ello. (Evitar saltos, multiplicaciones….)

**CÓDIGO EN ENSAMBLADOR** (no es necesario introducir aquí el código como captura de pantalla, ajustar el tamaño de la letra para que una instrucción no ocupe más de un renglón)**:**

**(PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR DONDE ESTÁ EL CÓDIGO EVALUADO, USE COLORES PARA DESTACAR LAS DIFERENCIAS)**

|  |  |  |  |
| --- | --- | --- | --- |
| daxpyO0.s | daxpyOs.s | daxpyO2.s | daxpyO3.s |
| 14fd: 66 48 0f 7e c0  movq %xmm0,%rax  1502: 48 89 45 b8  mov %rax,-0x48(%rbp)  1506: c7 45 80 00 00 00 00  movl $0x0,-0x80(%rbp)  150d: eb 3e  jmp 154d <main+0x304>  150f: 48 8b 45 a8  mov -0x58(%rbp),%rax  1513: 8b 55 80  mov -0x80(%rbp),%edx  1516: 48 63 d2  movslq %edx,%rdx  1519: f2 0f 10 04 d0  movsd (%rax,%rdx,8),%xmm0  151e: 66 0f 28 c8  movapd %xmm0,%xmm1  1522: f2 0f 59 4d b0  mulsd -0x50(%rbp),%xmm1  1527: 48 8b 45 98  mov -0x68(%rbp),%rax  152b: 8b 55 80  mov -0x80(%rbp),%edx  152e: 48 63 d2  movslq %edx,%rdx  1531: f2 0f 10 04 d0  movsd (%rax,%rdx,8),%xmm0  1536: f2 0f 58 c1  addsd %xmm1,%xmm0  153a: 48 8b 45 98  mov -0x68(%rbp),%rax  153e: 8b 55 80  mov -0x80(%rbp),%edx  1541: 48 63 d2  movslq %edx,%rdx  1544: f2 0f 11 04 d0  movsd %xmm0,(%rax,%rdx,8)  1549: 83 45 80 01  addl $0x1,-0x80(%rbp)  154d: 8b 45 80  mov -0x80(%rbp),%eax  1550: 3b 45 8c  cmp -0x74(%rbp),%eax  1553: 7c ba  jl 150f <main+0x2c6> | 12ad: f2 0f 10 4d c0  movsd -0x40(%rbp),%xmm1  12b2: 31 c0  xor %eax,%eax  12b4: f2 0f 11 45 c8  movsd %xmm0,-0x38(%rbp)  12b9: 39 c3  cmp %eax,%ebx  12bb: 7e 1c  jle 12d9 <main+0x179>  12bd: f2 41 0f 10 44 c5 00  movsd 0x0(%r13,%rax,8),%xmm0  12c4: f2 0f 59 c1  mulsd %xmm1,%xmm0  12c8: f2 41 0f 58 04 c4  addsd (%r12,%rax,8),%xmm0  12ce: f2 41 0f 11 04 c4  movsd %xmm0,(%r12,%rax,8)  12d4: 48 ff c0  inc %rax  12d7: eb e0  jmp 12b9 <main+0x159> | 1299: f2 0f 10 4d b0  movsd -0x50(%rbp),%xmm1  129e: 31 c0  xor %eax,%eax  12a0: f2 0f 11 45 b8  movsd %xmm0,-0x48(%rbp)  12a5: 0f 1f 00  nopl (%rax)  12a8: f2 41 0f 10 44 c5 00  Movsd 0x0(%r13,%rax,8),%xmm0  12af: f2 0f 59 c1  mulsd %xmm1,%xmm0  12b3: f2 0f 58 04 c3  addsd (%rbx,%rax,8),%xmm0  12b8: f2 0f 11 04 c3  movsd %xmm0,(%rbx,%rax,8)  12bd: 48 83 c0 01  add $0x1,%rax  12c1: 41 39 c6  cmp %eax,%r14d  12c4: 7f e2  jg 12a8 <main+0x148> | 1370: 45 85 ed  test %r13d,%r13d  1373: ba 01 00 00 00  mov $0x1,%edx  1378: 41 0f 4f d5  cmovg %r13d,%edx  137c: f2 0f 11 45 b8  movsd %xmm0,-0x48(%rbp)  1381: 7e 3f  jle 13c2 <main+0x262>  1383: 41 83 fd 01  cmp $0x1,%r13d  1387: 74 39  je 13c2 <main+0x262>  1389: f2 0f 10 4d b0  movsd -0x50(%rbp),%xmm1  138e: d1 ea  shr %edx  1390: 31 c0  xor %eax,%eax  1392: 48 c1 e2 04  shl $0x4,%rdx  1396: 66 0f 14 c9  unpcklpd %xmm1,%xmm1  139a: 66 0f 1f 44 00 00  nopw 0x0(%rax,%rax,1)  13a0: 66 41 0f 10 04 06  movupd (%r14,%rax,1),%xmm0  13a6: 66 41 0f 10 14 04  movupd (%r12,%rax,1),%xmm2  13ac: 66 0f 59 c1  mulpd %xmm1,%xmm0  13b0: 66 0f 58 c2  addpd %xmm2,%xmm0  13b4: 41 0f 11 04 04  movups %xmm0,(%r12,%rax,1)  13b9: 48 83 c0 10  add $0x10,%rax  13bd: 48 39 d0  cmp %rdx,%rax  13c0: 75 de  jne 13a0 <main+0x240> |

1. **~~(a)~~** ~~Paralizar con OpenMP en la CPU el código de la multiplicación resultante en el Ejercicio 1.(b). NOTA: usar para generar los valores aleatorios, por ejemplo, drand48\_r().~~

**~~(b)~~** ~~Calcular la ganancia en prestaciones que se obtiene en atcgrid4 para el máximo número de procesadores físicos con respecto al código inicial no optimizado del Ejercicio 1.(a) para dos tamaños de la matriz.~~

**~~(a) MULTIPLICACIÓN DE MATRICES PARALELO:~~**

**~~CAPTURA CÓDIGO FUENTE~~**~~: pmm-paralelo.c~~

|  |
| --- |
|  |

**~~(b) RESPUESTA~~**