|  |  |  |
| --- | --- | --- |
| **2º curso / 2º cuatr.**  **Grado Ing. Inform.** |  | **Arquitectura de Computadores (AC)**  **Cuaderno de prácticas.**  **Bloque Práctico 5. Optimización de código**  Estudiante (nombre y apellidos): Salvador Romero Cortés  Grupo de prácticas y profesor de prácticas: A1 Juan José Escobar Pérez  Fecha de entrega: 07/06/21  Fecha evaluación en clase: 01/06/21 |

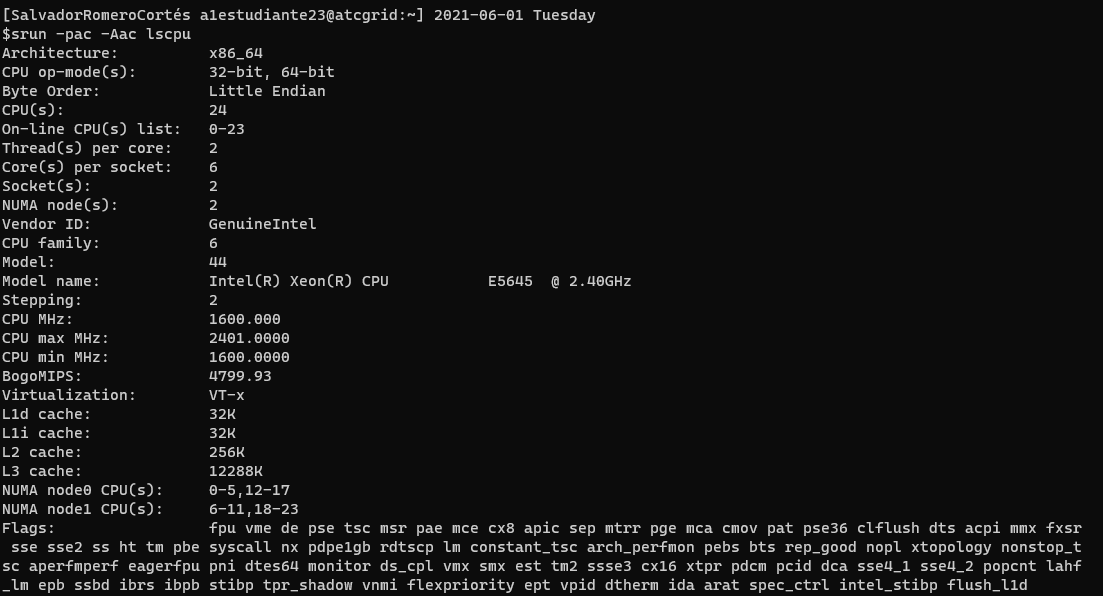
Antes de comenzar a realizar el trabajo de este cuaderno consultar el fichero con los normas de prácticas que se encuentra en SWAD

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): GenuineIntel Intel(R) Xeon(R) CPU E5645 @ 2.40GHz

Sistema operativo utilizado:CentOS Linux 7 (Core) Version 7

Versión de gcc utilizada: gcc version 9.2.0 (GCC)

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-:** desenrollar los bucles (4 iteraciones)

**Modificación B) –explicación-:** intercambiar las variables j y k

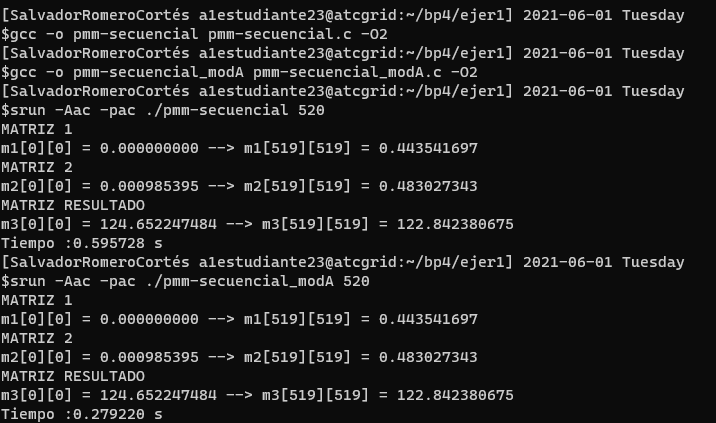
**...**

**CÓDIGOS FUENTE MODIFICACIONES**

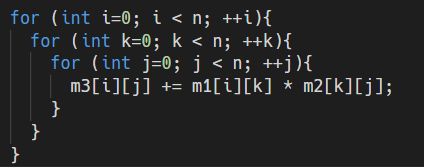
**A) Captura de** pmm-secuencial-modificado\_A.c

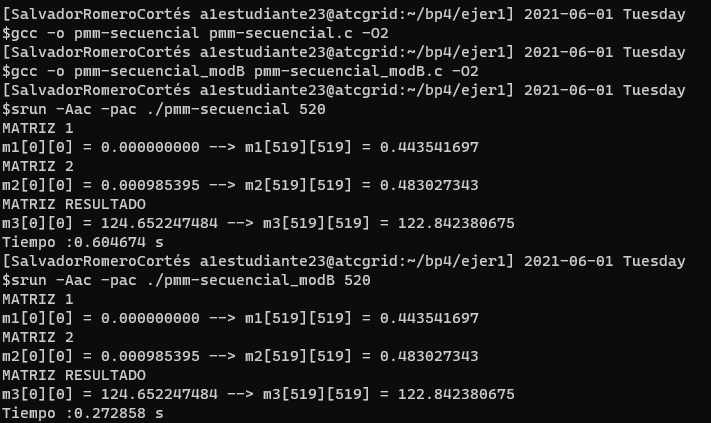
|  |
| --- |
| Código distinto: |

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

****

**B) ...**

****

****

**TIEMPOS:**

|  |  |  |
| --- | --- | --- |
| **Modificación** | **Breve descripción de las modificaciones** | **-O2** |
| Sin modificar |  | 0.595728 |
| Modificación A) | Desenrollado de bucle con 4 iteraciones | 0.279220 |
| Modificación B) | Intercambiar las variables j y k | 0.272858 |
| ... |  |  |
|  |  |  |

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

Sobre la primera modificación: al desenrollar el bucle con 4 iteraciones, si usamos un tamaño que no sea múltiplo de 4 se producirá un segmentation fault o algún error de memoria similar por lo que esta optimización sólo la podemos aplicar con un tamaño que sepamos que sea múltiplo de 4. La mejora de tiempo ocurre porque se reduce el número de saltos y aumenta la oportunidad de encontrar instrucciones independientes.

Sobre la segunda modificación: intercambiamos las variables j y k para mejorar la localidad espacial y de esta manera los datos en memoria están más cerca entre los distintos accesos, haciendo más eficiente el uso de caché puesto que si la localización espacial es buena no habrá que traer más datos de memoria a caché.

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-:** cambiar los productos por desplazamientos de bits y usar un único bucle en lugar de dos.

**Modificación B) –explicación-:** cambiamos el if del final por el operador ternario y desenrollamos los bucles internos que modifican X1 y X2

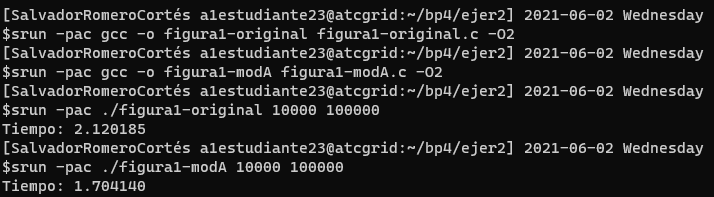
**...**

**CÓDIGOS FUENTE MODIFICACIONES**

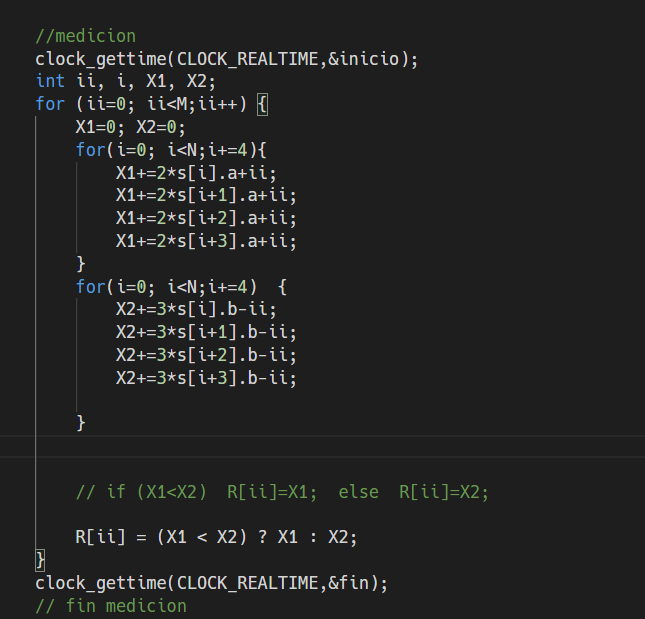
**A) Captura** figura1-modificado\_A.c

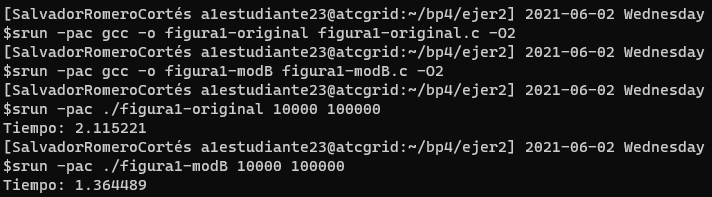
|  |
| --- |
| Código que cambia: |

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

****

**B) ...**

****

****

**TIEMPOS:**

|  |  |  |
| --- | --- | --- |
| **Modificación** | **Breve descripción de las modificaciones** | **-O2** |
| Sin modificar |  | 2.120185 |
| Modificación A) | Sustituir operadores \* por << y unificar los dos bucles | 1.704140 |
| Modificación B) | Cambio de if por operador ternario y desenrollo de bucle | 1.364489 |
| ... |  |  |
|  |  |  |

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

Sobre la primera modificación:

La verdadera mejora es provocada por unificar los bucles puesto que estaremos haciendo N ejecuciones en lugar de 2N ejecuciones. En este caso, sólo cambiar los operadores no mejora mucho al provocar más accesos a la memoria principal.

Sobre la segunda modificación:

Vemos como desenrollar los bucles sigue siendo la mejor opción para ganar velocidad de ejecución. Las justificación se explicó en el ejercicio anterior cuando también desenrollamos ese bucle.

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];

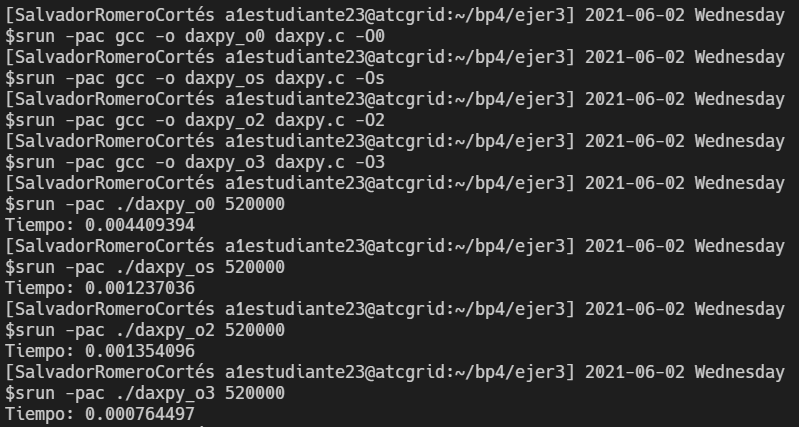
Generar los programas en ensamblador para cada una de las siguientes opciones de optimización del compilador: -O0, -Os, -O2, -O3. Explique las diferencias que se observan en el código justificando al mismo tiempo las mejoras en velocidad que acarrean. Incorporar los códigos al cuaderno de prácticas y 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=520000** | **-O0** | **-Os** | **-O2** | **-O3** |
| 0.004409394 | 0.001237036 | 0.001354096 | 0.000764497 |

**CAPTURAS DE PANTALLA** (que muestren la compilación y que el resultado es correcto)**:**

****

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

He intentado dejar en negro las partes que son comunes entre al menos dos versiones del código ensamblador y en colores lo que cambia según el nivel de optimización.

Vemos como apenas hay diferencias entre Os y O2. Esto ocurre porque Os ya activa todos los flags de optimización del nivel O2. Se puede ver también como el código de O3 es ligeramente más largo que el de Os y O2.

También notamos que aparecen más “números mágicos” sin ningún nivel de optimización, así como el código más largo.

Además podemos ver en los tiempos de ejecución que las optimizaciones funcionan correctamente: sin optimizar es el que más tarda, Os y O2 tardan casi el mismo tiempo (la diferencia es mínima y es posible que se eliminara si ejecutáramos varias veces ambos y hiciéramos la media) y finalmente, O3 es el que menos tarda con bastante diferencia.

**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 |
| movl    $0, -56(%rbp)      jmp .L5  .L6:      movq    -88(%rbp), %rax      movl    -56(%rbp), %edx      movslq  %edx, %rdx      movsd   (%rax,%rdx,8), %xmm0      movapd  %xmm0, %xmm1      mulsd   -72(%rbp), %xmm1      movq    -104(%rbp), %rax      movl    -56(%rbp), %edx      movslq  %edx, %rdx      movsd   (%rax,%rdx,8), %xmm0      addsd   %xmm1, %xmm0      movq    -104(%rbp), %rax      movl    -56(%rbp), %edx      movslq  %edx, %rdx      movsd   %xmm0, (%rax,%rdx,8)      addl    $1, -56(%rbp)  .L5:      movl    -56(%rbp), %eax      cmpl    -52(%rbp), %eax      jl  .L6      leaq    -144(%rbp), %rax      movq    %rax, %rsi      movl    $0, %edi | movsd   .LC1(%rip), %xmm1      xorl    %eax, %eax  .L5:      cmpl    %eax, %ebx      jle .L10      movsd   0(%r13,%rax,8), %xmm0      mulsd   %xmm1, %xmm0      addsd   (%r14,%rax,8), %xmm0      movsd   %xmm0, (%r14,%rax,8)      incq    %rax      jmp .L5  .L10:      leaq    -48(%rbp), %rsi      xorl    %edi, %edi | movsd   .LC1(%rip), %xmm1      xorl    %eax, %eax      .p2align 4,,10      .p2align 3  .L6:      movsd   (%r12,%rax,8), %xmm0      movq    %rax, %rdx      mulsd   %xmm1, %xmm0      addsd   (%r14,%rax,8), %xmm0      movsd   %xmm0, (%r14,%rax,8)      addq    $1, %rax      cmpq    %r13, %rdx      jne .L6  .L7:      leaq    -48(%rbp), %rsi      xorl    %edi, %edi | cmpl    $1, %r12d      je  .L8      shrl    %r12d      movapd  .LC1(%rip), %xmm1      xorl    %edx, %edx      salq    $4, %r12      .p2align 4,,10      .p2align 3  .L6:      movupd  0(%r13,%rdx), %xmm0      movupd  (%r15,%rdx), %xmm2      mulpd   %xmm1, %xmm0      addpd   %xmm2, %xmm0      movups  %xmm0, (%r15,%rdx)      addq    $16, %rdx      cmpq    %rdx, %r12      jne .L6  .L8:      leaq    -64(%rbp), %rsi      xorl    %edi, %edi |

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.

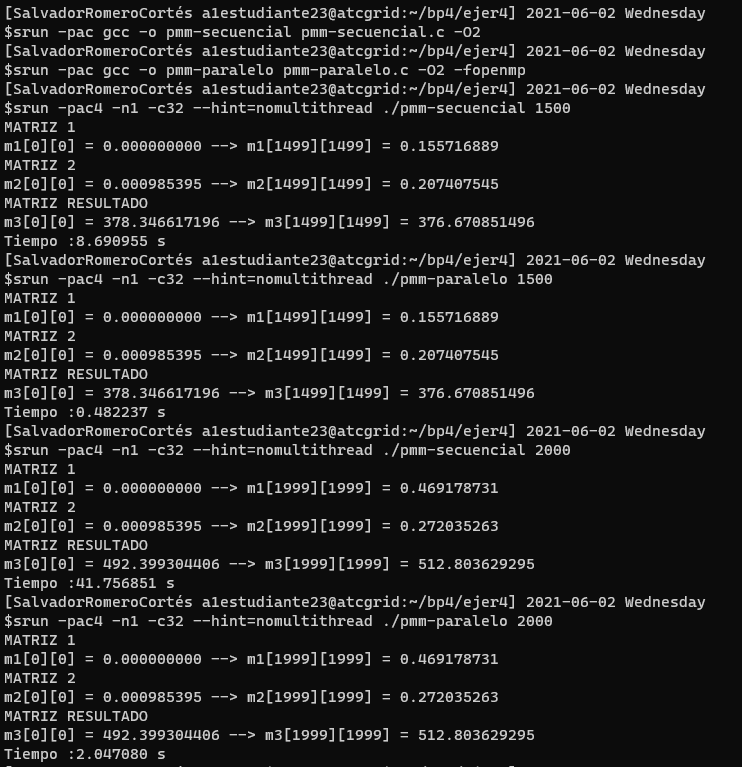
**(a) MULTIPLICACIÓN DE MATRICES PARALELO:**

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

|  |
| --- |
|  |

**(b) RESPUESTA**

Si ejecutamos ambos programas con 32 cores en el atcgrid4 obtenemos los siguientes resultados:



Vemos que la ganancia es bastante notable. Si la calculamos:

Tamaño 1000:

Ganancia = Tiempo\_Secuencial / Tiempo\_Paralelo = 8.690955 / 0.482237 = 18.0222

Tamaño 2000:

Ganancia = Tiempo\_Secuencial / Tiempo\_Paralelo = 41.756851 / 2.047080 = 20.3983