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:

```
[SalvadorRomeroCortés a1estudiante23@atcgrid:~] 2021-06-01 Tuesday
$srun -pac -Aac lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU           E5645  @ 2.40GHz
Stepping:              2
CPU MHz:               1600.000
CPU max MHz:           2401.0000
CPU min MHz:           1600.0000
BogoMIPS:              4799.93
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
 sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_t
sc aperfmperf eagerfpu pni dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 popcnt lahf
_lm epb ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid dtherm ida arat spec_ctrl intel_stibp flush_l1d
```

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

```c
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h>  // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
#define VECTOR_DYNAMIC  // descomentar para que los vectores sean variables ...
    // dinámicas (memoria reutilizable durante la ejecución)

#ifdef VECTOR_GLOBAL
#define MAX 33554432  //=2^25
//#define MAX 4294967295  //=2^32 -1

double m1[MAX], m2[MAX], m3[MAX];
#endif
int main(int argc, char** argv){

  int i;

  struct timespec inicio,fin; double tiempo; //para tiempo de ejecución

  //Leer argumento de entrada (nº de componentes del vector)
  if (argc<2){
    printf("Faltan nº componentes del vector\n");
    exit(-1);
  }

  unsigned int n = atoi(argv[1]); // Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)
  //printf("Tamaño Vectores:%u (%u B)\n",N, sizeof(unsigned int));
  #ifdef VECTOR_LOCAL
  double m1[N], m2[N], m3[N];   // Tamaño variable local en tiempo de ejecución ...
      // disponible en C a partir de C99
  #endif
  #ifdef VECTOR_GLOBAL
  if (N>MAX) N=MAX;
  #endif
  #ifdef VECTOR_DYNAMIC
  double **m1, **m2, **m3;
  m1 = (double**) malloc(n*sizeof(double*));// malloc necesita el tamaño en bytes
  m2 = (double**) malloc(n*sizeof(double*));
  m3 = (double**) malloc(n*sizeof(double*));
  if ((m1 == NULL) || (m2 == NULL) || (m3 == NULL)) {
    printf("No hay suficiente espacio para los vectores \n");
    exit(-2);
  }
  #endif

  //Reserva de memoria de las matrices
  for (int i=0; i < n; ++i){
    m1[i] = (double *) malloc (n*sizeof(double));
    m2[i] = (double *) malloc (n*sizeof(double));
    m3[i] = (double *) malloc (n*sizeof(double));
  }
```

```
//inicializacion de los datos
if (n <= 8){
  for (int i=0; i < n; i++){
    for (int j=0; j < n; j++){
      m1[i][j] = n * 0.1 + i * 0.1;
      m2[i][j] = n * 0.1 - i * 0.1;
      m3[i][j] = 0;
    }
  }
} else{
  for (int i=0; i < n; ++i){
    for (int j=0; j < n; ++j){
      m1[i][j] = drand48();
      m2[i][j] = drand48();
      m3[i][j] = 0;
    }
  }
}

//calculo del producto
clock_gettime(CLOCK_REALTIME,&inicio);

for (int i=0; i < n; ++i){
  for (int j=0; j < n; ++j){
    for (int k=0; k < n; ++k){
      m3[i][j] += m1[i][k] * m2[k][j];
    }
  }
}

clock_gettime(CLOCK_REALTIME,&fin);
tiempo=(double) (fin.tv_sec-inicio.tv_sec)+
    (double) ((fin.tv_nsec-inicio.tv_nsec)/(1.e+9));

//Imprimir resultado de la suma y el tiempo de ejecución

if (n <= 10){
  printf("MATRIZ 1\n");
  for (int i=0; i < n; ++i){
    for (int j=0; j < n; ++j)
      printf("%11.9f - ", m1[i][j]);
    printf("\n");
  }

  printf("MATRIZ 2\n");
  for (int i=0; i < n; ++i){
    for (int j=0; j < n; ++j)
      printf("%11.9f - ", m2[i][j]);
    printf("\n");
  }
```

```
    printf("MATRIZ RESULTADO\n");
    for (int i=0; i < n; ++i){
      for (int j=0; j < n; ++j)
        printf("%11.9f - ", m3[i][j]);
      printf("\n");
    }

    printf("Tiempo: %f s\n", tiempo);
  } else{
    printf("MATRIZ 1\n");
    printf("m1[0][0] = %11.9f --> m1[%d][%d] = %11.9f\n", m1[0][0], n-1, n-1, m1[n-1][n-1]);

    printf("MATRIZ 2\n");
    printf("m2[0][0] = %11.9f --> m2[%d][%d] = %11.9f\n", m2[0][0], n-1, n-1, m2[n-1][n-1]);

    printf("MATRIZ RESULTADO\n");
    printf("m3[0][0] = %11.9f --> m3[%d][%d] = %11.9f\n", m3[0][0], n-1, n-1, m3[n-1][n-1]);

    printf("Tiempo :%f s\n", tiempo);
  }


  #ifdef VECTOR_DYNAMIC
  for (int i=0; i < n; i++){
    free(m1[i]);
    free(m2[i]);
    free(m3[i]);
  }

  free(m1); free(m2); free(m3);
  #endif
  return 0;
}
```

(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:

```
for (int i=0; i < n; ++i){
  for (int j=0; j < n; j+=4){
    for (int k=0; k < n; ++k){
      m3[i][j] += m1[i][k] * m2[k][j];
      m3[i][j+1] += m1[i][k] * m2[k][j+1];
      m3[i][j+2] += m1[i][k] * m2[k][j+2];
      m3[i][j+3] += m1[i][k] * m2[k][j+3];
    }
  }
}
```

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

```
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer1] 2021-06-01 Tuesday
$gcc -o pmm-secuencial pmm-secuencial.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer1] 2021-06-01 Tuesday
$gcc -o pmm-secuencial_modA pmm-secuencial_modA.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer1] 2021-06-01 Tuesday
$srun -Aac -pac ./pmm-secuencial 520
MATRIZ 1
m1[0][0] = 0.000000000 --> m1[519][519] = 0.443541697
MATRIZ 2
m2[0][0] = 0.000985395 --> m2[519][519] = 0.483027343
MATRIZ RESULTADO
m3[0][0] = 124.652247484 --> m3[519][519] = 122.842380675
Tiempo :0.595728 s
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer1] 2021-06-01 Tuesday
$srun -Aac -pac ./pmm-secuencial_modA 520
MATRIZ 1
m1[0][0] = 0.000000000 --> m1[519][519] = 0.443541697
MATRIZ 2
m2[0][0] = 0.000985395 --> m2[519][519] = 0.483027343
MATRIZ RESULTADO
m3[0][0] = 124.652247484 --> m3[519][519] = 122.842380675
Tiempo :0.279220 s
```

B) ...

```
for (int i=0; i < n; ++i){
  for (int k=0; k < n; ++k){
    for (int j=0; j < n; ++j){
      m3[i][j] += m1[i][k] * m2[k][j];
    }
  }
}
```

```
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer1] 2021-06-01 Tuesday
$gcc -o pmm-secuencial pmm-secuencial.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer1] 2021-06-01 Tuesday
$gcc -o pmm-secuencial_modB pmm-secuencial_modB.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer1] 2021-06-01 Tuesday
$srun -Aac -pac ./pmm-secuencial 520
MATRIZ 1
m1[0][0] = 0.000000000 --> m1[519][519] = 0.443541697
MATRIZ 2
m2[0][0] = 0.000985395 --> m2[519][519] = 0.483027343
MATRIZ RESULTADO
m3[0][0] = 124.652247484 --> m3[519][519] = 122.842380675
Tiempo :0.604674 s
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer1] 2021-06-01 Tuesday
$srun -Aac -pac ./pmm-secuencial_modB 520
MATRIZ 1
m1[0][0] = 0.000000000 --> m1[519][519] = 0.443541697
MATRIZ 2
m2[0][0] = 0.000985395 --> m2[519][519] = 0.483027343
MATRIZ RESULTADO
m3[0][0] = 124.652247484 --> m3[519][519] = 122.842380675
Tiempo :0.272858 s
```

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.

2. **(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

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

struct tipo {
        int a;
        int b;
};

int main(int argc, char ** argv)
{
    //control de errores
    if (argc != 3){
        printf("Faltan argumentos: N y M\n");
        exit(-1);
    }

    // declaraciones de variables
    int N = atoi(argv[1]);
    int M = atoi(argv[2]);
    struct tipo s[N];
    int R[M];
    struct timespec inicio,fin; double tiempo;

    // inicializacion
    if (N <= 8){
        for (int i=0; i < N; i++){
            s[i].a = N * 0.1 + i * 0.1;
            s[i].b = N * 0.1 - i * 0.1;

        }
    } else{
        for (int i=0; i < N; i++){
            s[i].a = drand48();
            s[i].b = drand48();

        }
    }
    for (int i=0; i < M; i++){
        R[i] = 0;
    }


    //medicion
    clock_gettime(CLOCK_REALTIME,&inicio);
    int ii, i, X1, X2;
    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;
    }
    clock_gettime(CLOCK_REALTIME,&fin);
    // fin medicion
    tiempo = (double) (fin.tv_sec-inicio.tv_sec)+
      (double) ((fin.tv_nsec-inicio.tv_nsec)/(1.e+9));

    printf("Tiempo: %f\n", tiempo);
}
```

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:

```
//medicion
clock_gettime(CLOCK_REALTIME,&inicio);
int ii, i, X1, X2;
for (ii=0; ii<M;ii++) {
    X1=0; X2=0;
    for(i=0; i<N;i++) {
        X1+=s[i].a << 1 +ii; X2+=s[i].b << 1  + s[i].b -ii;
    }

    if (X1<X2)  R[ii]=X1;  else  R[ii]=X2;

}
clock_gettime(CLOCK_REALTIME,&fin);
// fin medicion
```

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

```
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer2] 2021-06-02 Wednesday
$srun -pac gcc -o figura1-original figura1-original.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer2] 2021-06-02 Wednesday
$srun -pac gcc -o figura1-modA figura1-modA.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer2] 2021-06-02 Wednesday
$srun -pac ./figura1-original 10000 100000
Tiempo: 2.120185
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer2] 2021-06-02 Wednesday
$srun -pac ./figura1-modA 10000 100000
Tiempo: 1.704140
```

B) ...

```
//medicion
clock_gettime(CLOCK_REALTIME,&inicio);
int ii, i, X1, X2;
for (ii=0; ii<M;ii++) {
    X1=0; X2=0;
    for(i=0; i<N;i+=4){
        X1+=2*s[i].a+ii;
        X1+=2*s[i+1].a+ii;
        X1+=2*s[i+2].a+ii;
        X1+=2*s[i+3].a+ii;
    }
    for(i=0; i<N;i+=4)  {
        X2+=3*s[i].b-ii;
        X2+=3*s[i+1].b-ii;
        X2+=3*s[i+2].b-ii;
        X2+=3*s[i+3].b-ii;

    }


    // if (X1<X2)  R[ii]=X1;  else  R[ii]=X2;

    R[ii] = (X1 < X2) ? X1 : X2;
}
clock_gettime(CLOCK_REALTIME,&fin);
// fin medicion
```

```
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer2] 2021-06-02 Wednesday
$srun -pac gcc -o figura1-original figura1-original.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer2] 2021-06-02 Wednesday
$srun -pac gcc -o figura1-modB figura1-modB.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer2] 2021-06-02 Wednesday
$srun -pac ./figura1-original 10000 100000
Tiempo: 2.115221
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer2] 2021-06-02 Wednesday
$srun -pac ./figura1-modB 10000 100000
Tiempo: 1.364489
```

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.

3. 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char ** argv){
    if (argc != 2){
        printf("Error en la llamda. Hace falta el parámetro N (tamaño de vector)\n");
        exit(EXIT_FAILURE);
    }

    struct timespec inicio, fin;
    double tiempo;

    const int N = atoi(argv[1]);
    double a = 13.42;
    double x[N];
    double y[N];

    // inicalizacion de los vectores
    for (int i=0; i < N; ++i){
        srand48(time(NULL));
        x[i] = drand48();
        srand48(time(NULL));;
        y[i] = drand48();
    }
    clock_gettime(CLOCK_REALTIME, &inicio);
    for (int i=0;i<N;i++) y[i]= a*x[i] + y[i];
    clock_gettime(CLOCK_REALTIME, &fin);

    tiempo = (double) (fin.tv_sec-inicio.tv_sec)+
       (double) ((fin.tv_nsec-inicio.tv_nsec)/(1.e+9));
    printf("Tiempo: %11.9f\n",tiempo);
}
```

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)**:**

```
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer3] 2021-06-02 Wednesday
$srun -pac gcc -o daxpy_o0 daxpy.c -O0
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer3] 2021-06-02 Wednesday
$srun -pac gcc -o daxpy_os daxpy.c -Os
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer3] 2021-06-02 Wednesday
$srun -pac gcc -o daxpy_o2 daxpy.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer3] 2021-06-02 Wednesday
$srun -pac gcc -o daxpy_o3 daxpy.c -O3
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer3] 2021-06-02 Wednesday
$srun -pac ./daxpy_o0 520000
Tiempo: 0.004409394
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer3] 2021-06-02 Wednesday
$srun -pac ./daxpy_os 520000
Tiempo: 0.001237036
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer3] 2021-06-02 Wednesday
$srun -pac ./daxpy_o2 520000
Tiempo: 0.001354096
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer3] 2021-06-02 Wednesday
$srun -pac ./daxpy_o3 520000
Tiempo: 0.000764497
```

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 esta se eliminara si ejecutáramos varias veces ambos e 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

```
    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)
```

daxpyOs.s

```
    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
```

daxpyO2.s

```
    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
```

daxpyO3.s

```
    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
```

.L5:
movl -56(%rbp), %eax
cmpl -52(%rbp), %eax
jl .L6
leaq -144(%rbp), %rax
movq %rax, %rsi
movl $0, %edi
			xorl %edi, %edi

4. **(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

```c
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h>  // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
#include <omp.h>
#define VECTOR_DYNAMIC  // descomentar para que los vectores sean variables ...
    // dinámicas (memoria reutilizable durante la ejecución)

#ifdef VECTOR_GLOBAL
#define MAX 33554432  //=2^25
//#define MAX 4294967295  //=2^32 -1

double m1[MAX], m2[MAX], m3[MAX];
#endif
int main(int argc, char** argv){

  int i;

  double inicio, tiempo; //para tiempo de ejecución

  //Leer argumento de entrada (nº de componentes del vector)
  if (argc<2){
    printf("Faltan nº componentes del vector\n");
    exit(-1);
  }

  unsigned int n = atoi(argv[1]); // Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)
  //printf("Tamaño Vectores:%u (%u B)\n",N, sizeof(unsigned int));
  #ifdef VECTOR_LOCAL
  double m1[N], m2[N], m3[N];   // Tamaño variable local en tiempo de ejecución ...
      // disponible en C a partir de C99
  #endif
  #ifdef VECTOR_GLOBAL
  if (N>MAX) N=MAX;
  #endif
  #ifdef VECTOR_DYNAMIC
  double **m1, **m2, **m3;
  m1 = (double**) malloc(n*sizeof(double*));// malloc necesita el tamaño en bytes
  m2 = (double**) malloc(n*sizeof(double*));
  m3 = (double**) malloc(n*sizeof(double*));
  if ((m1 == NULL) || (m2 == NULL) || (m3 == NULL)) {
    printf("No hay suficiente espacio para los vectores \n");
    exit(-2);
  }
  #endif

  //Reserva de memoria de las matrices
  for (int i=0; i < n; ++i){
    m1[i] = (double *) malloc (n*sizeof(double));
    m2[i] = (double *) malloc (n*sizeof(double));
    m3[i] = (double *) malloc (n*sizeof(double));
  }
```

```
//inicializacion de los datos
if (n <= 8){
  for (int i=0; i < n; i++){
    for (int j=0; j < n; j++){
      m1[i][j] = n * 0.1 + i * 0.1;
      m2[i][j] = n * 0.1 - i * 0.1;
      m3[i][j] = 0;
    }
  }
} else{
  for (int i=0; i < n; ++i){
    for (int j=0; j < n; ++j){
      m1[i][j] = drand48();
      m2[i][j] = drand48();
      m3[i][j] = 0;
    }
  }
}

//calculo del producto
inicio = omp_get_wtime();
for (int i=0; i < n; ++i){
  #pragma omp parallel for
  for (int k=0; k < n; ++k){
    for (int j=0; j < n; ++j){
      m3[i][j] += m1[i][k] * m2[k][j];
    }
  }
}

tiempo = omp_get_wtime() - inicio ;

//Imprimir resultado de la suma y el tiempo de ejecución

if (n <= 10){
  printf("MATRIZ 1\n");
  for (int i=0; i < n; ++i){
    for (int j=0; j < n; ++j)
      printf("%11.9f - ", m1[i][j]);
    printf("\n");
  }

  printf("MATRIZ 2\n");
  for (int i=0; i < n; ++i){
    for (int j=0; j < n; ++j)
      printf("%11.9f - ", m2[i][j]);
    printf("\n");
  }

  printf("MATRIZ RESULTADO\n");
  for (int i=0; i < n; ++i){
    for (int j=0; j < n; ++j)
      printf("%11.9f - ", m3[i][j]);
    printf("\n");
  }

  printf("Tiempo: %f s\n", tiempo);
} else{
```

```
    printf("MATRIZ 1\n");
    printf("m1[0][0] = %11.9f --> m1[%d][%d] = %11.9f\n", m1[0][0], n-1, n-1, m1[n-1][n-1]);

    printf("MATRIZ 2\n");
    printf("m2[0][0] = %11.9f --> m2[%d][%d] = %11.9f\n", m2[0][0], n-1, n-1, m2[n-1][n-1]);

    printf("MATRIZ RESULTADO\n");
    printf("m3[0][0] = %11.9f --> m3[%d][%d] = %11.9f\n", m3[0][0], n-1, n-1, m3[n-1][n-1]);

    printf("Tiempo :%f s\n", tiempo);
  }


  #ifdef VECTOR_DYNAMIC
  for (int i=0; i < n; i++){
    free(m1[i]);
    free(m2[i]);
    free(m3[i]);
  }

  free(m1); free(m2); free(m3);
  #endif
  return 0;
}
```

(b) RESPUESTA

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

```
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer4] 2021-06-04 Friday
$srun -pac gcc -o pmm-secuencial pmm-secuencial.c -O2
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer4] 2021-06-04 Friday
$srun -pac gcc -o pmm-paralelo pmm-paralelo.c -O2 -fopenmp
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer4] 2021-06-04 Friday
$srun -pac4 -n1 -c32 --hint=nomultithread ./pmm-secuencial 1500
MATRIZ 1
m1[0][0] = 0.000000000 --> m1[1499][1499] = 0.155716889
MATRIZ 2
m2[0][0] = 0.000985395 --> m2[1499][1499] = 0.207407545
MATRIZ RESULTADO
m3[0][0] = 378.346617196 --> m3[1499][1499] = 376.670851496
Tiempo :8.744559 s
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer4] 2021-06-04 Friday
$srun -pac4 -n1 -c32 --hint=nomultithread ./pmm-paralelo 1500
MATRIZ 1
m1[0][0] = 0.000000000 --> m1[1499][1499] = 0.155716889
MATRIZ 2
m2[0][0] = 0.000985395 --> m2[1499][1499] = 0.207407545
MATRIZ RESULTADO
m3[0][0] = 227.462069459 --> m3[1499][1499] = 160.879390866
Tiempo :2.893818 s
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer4] 2021-06-04 Friday
$srun -pac4 -n1 -c32 --hint=nomultithread ./pmm-secuencial 2000
MATRIZ 1
m1[0][0] = 0.000000000 --> m1[1999][1999] = 0.469178731
MATRIZ 2
m2[0][0] = 0.000985395 --> m2[1999][1999] = 0.272035263
MATRIZ RESULTADO
m3[0][0] = 492.399304406 --> m3[1999][1999] = 512.803629295
Tiempo :38.304383 s
[SalvadorRomeroCortés a1estudiante23@atcgrid:~/bp4/ejer4] 2021-06-04 Friday
$srun -pac4 -n1 -c32 --hint=nomultithread ./pmm-paralelo 2000
MATRIZ 1
m1[0][0] = 0.000000000 --> m1[1999][1999] = 0.469178731
MATRIZ 2
m2[0][0] = 0.000985395 --> m2[1999][1999] = 0.272035263
MATRIZ RESULTADO
m3[0][0] = 330.315005410 --> m3[1999][1999] = 262.016693348
Tiempo :6.530322 s
```

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

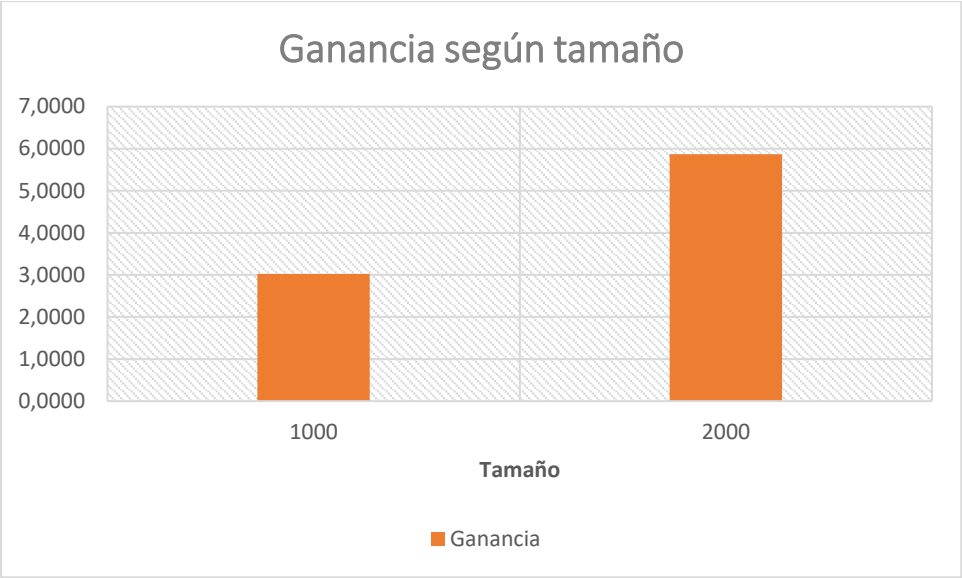
Tamaño 1000:

Ganancia = Tiempo_Secuencial / Tiempo_Paralelo = 8.744559 / 2.893818 = 3.0218

Tamaño 2000:

Ganancia = Tiempo_Secuencial / Tiempo_Paralelo = 38.304383 / 6.530322 = 5.8656

Para visualizalo mejor, aquí temenos un pequeño gráfico:


Ganancia según tamaño
7,0000
6,0000
5,0000
4,0000
3,0000
2,0000
1,0000
0,0000
1000
2000
Tamaño
Ganancia