

2º curso / 2º cuatr.

Grado Ing. Inform.

Doble Grado Ing. Inform. y Mat.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.


Bloque Práctico 4. Optimización de código

Estudiante (nombre y apellidos):Carlos de la Torre

Grupo de prácticas:A2

Fecha de entrega: 10/6

Fecha evaluación en clase: 11/6



Versión de gcc utilizada: gcc versión 4.8.2 20131212 (Red Hat 4.8.2-7) (GCC)

Adjunte en un fichero el contenido del fichero /proc/cpuinfo de la máquina en la que ha tomado las medidas:

```
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 37
model name     : Intel(R) Core(TM) i5 CPU           M 430  @ 2.27GHz
stepping       : 2
microcode      : 0xe
cpu MHz        : 1199.000
cache size     : 3072 KB
physical id    : 0
siblings       : 4
core id        : 0
cpu cores      : 2
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 11
wp             : yes
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 rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl
xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_cpl vmx est tm2
ssse3 cx16 xtpr pdcm sse4_1 sse4_2 popcnt lahf_lm ida arat dtherm
tptr_shadow vnmi flexpriority ept vpid
bogomips       : 4522.63
clflush size   : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
```

| | | | | |
|----------------|-----------------------|-----------------------|-----------------------|------------------------|
| Cache: | L1 data | L1 instruction | L2 | L3 |
| Size: | 2 x 32 KB | 2 x 32 KB | 2 x 256 KB | 3 MB |
| Associativity: | 8-way set associative | 4-way set associative | 8-way set associative | 12-way set associative |

Claro esta esto por 4 procesadores

1. Para el núcleo que se muestra en la Figura 1, y para un programa que implemente la multiplicación de matrices:
 - a. Modifique el código C para reducir el tiempo de ejecución del mismo. Justifique los tiempos obtenidos a partir de la modificación realizada.
 - b. Genere los programas en ensamblador para los programas modificados obtenidos en el punto anterior considerando las distintas opciones de optimización del compilador (-O1, -O2,...). Compare los tiempos de ejecución de las versiones de código ejecutable obtenidas con las distintas opciones de optimización y explique las diferencias en tiempo a partir de las características de dichos códigos.
 - c. (Ejercicio EXTRA) Intente mejorar los resultados obtenidos transformando el código ensamblador del programa para el que se han conseguido las mejores prestaciones de tiempo

```

struct {
    int a;
    int b;
} s[5000];

main()
{
    ...
    for (ii=1; ii<=40000;ii++) {
        for(i=0; i<5000;i++) X1=2*s[i].a+ii;
        for(i=0; i<5000;i++) X2=3*s[i].b-ii;

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

```

Figura 1: Núcleo de programa en C para el ejercicio 1.

A) MULTIPLICACIÓN DE MATRICES:**CÓDIGO FUENTE:** pmm-secuencial-modificado.c

```

/*
 * pmm-modificado.c
 *
 * Created on: 25/05/2014
 * Author: Carlos de la Torre
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define PRINT_ALL_MIN 15
// Ponemos que los elementos mínimos para que se
// impriman todos los valores de la matriz sea 15
#define DEBUGMODE 0
// con esta definición nos aseguramos que solo
// salgan las cifras de tiempo en cada ejecución
// así de esa manera es mas fácil realizar el
// estudio empírico del programa

void mejoras(int **M1, int **M2, int **MR, int N);

int main(int argc, char* argv[]) {
    int f,c,N;
    double tr;
    struct timespec t1, t2;
    int boolImprime = 0;

```

```

        switch (argc){ // coneste switch nos aseguramos de que la entrada de parametros
sea correcta
        case 1:
            printf("Faltan las filas/columnas de la Matrices\n");
            printf("\nUso: %s [numero]\n",argv[0]);
            printf("\nDonde numero es el tamaño de las filas y las columnas
de la matrices\n");
            exit(-1);
            break;
        case 2:
            N = atoi(argv[1]); // Este sera el tamaño del vector y de las
filas/columnas de la matriz
            break;
        case 3:
            N = atoi(argv[1]); // Este sera el tamaño del vector y de las
filas/columnas de la matriz
            if (atoi(argv[2])==1)
                boolImprime = 1;
            break;
        default:
            printf("La cantidad de parametros es incorrecta\n");
            exit(-1);
            break;
    }

#ifdef MEJORA_MEM
    // dos temporales para poder alinear la memoria *tempA, *tempB;
    const int LINEA_CACHE = 64;
    const int TAM_BITS = 8192;

    int **M1, **M2, **MR;
    M1 = (int**) malloc(sizeof(int*) * N + LINEA_CACHE-1);
    for (f = 0; f < N; f++)
        M1[f] = (int*) malloc(sizeof(int*) * N + LINEA_CACHE-1);
    M2 = (int**) malloc(sizeof(int*) * N + LINEA_CACHE-1);
    for (f = 0; f < N; f++)
        M2[f] = (int*) malloc(sizeof(int*) * N + LINEA_CACHE-1);
    MR = (int**) malloc(sizeof(int*) * N + LINEA_CACHE-1);
    for (f = 0; f < N; f++)
        MR[f] = (int*) malloc(sizeof(int*) * N + LINEA_CACHE-1);
    // los 63 es tamaño línea de cache
    int *tempM1A = (int *)(((int)M1+LINEA_CACHE-1)&~(LINEA_CACHE-1));
    int *tempM2A = (int *)(((int)M2+LINEA_CACHE-1)&~(LINEA_CACHE-1));
    int *tempMRA = (int *)(((int)MR+LINEA_CACHE-1)&~(LINEA_CACHE-1));
    // los 8192 son 32KB de tamaño cache nivel 1
    // por 2 memorias que tiene mi procesador
    // dividido entre 8 vias se quedan 8192
    int *tempM1B = (int *)((((int)M1+LINEA_CACHE-1)&~(LINEA_CACHE-1))
+TAM_BITS+LINEA_CACHE);
    int *tempM2B = (int *)((((int)M2+LINEA_CACHE-1)&~(LINEA_CACHE-1))
+TAM_BITS+LINEA_CACHE);
    int *tempMRB = (int *)((((int)MR+LINEA_CACHE-1)&~(LINEA_CACHE-1))
+TAM_BITS+LINEA_CACHE);

    if ((M1 == NULL) || (M2 == NULL) || (MR == NULL)) {
        printf("Error en la reserva de espacio para los Vectores o MatrizTri\n");
        exit(-2);
    }

#else
    /* aquí se hace la reserva de memoria normal con
    * malloc y se devuelve como puntero
    */
    int **M1, **M2, **MR;
    M1 = (int**) malloc(sizeof(int*) * N);
    for (f = 0; f < N; f++)
        M1[f] = (int*) malloc(sizeof(int*) * N);
    M2 = (int**) malloc(sizeof(int*) * N);
    for (f = 0; f < N; f++)
        M2[f] = (int*) malloc(sizeof(int*) * N);
    MR = (int**) malloc(sizeof(int*) * N);
    for (f = 0; f < N; f++)
        MR[f] = (int*) malloc(sizeof(int*) * N);
    if ((M1 == NULL) || (M2 == NULL) || (MR == NULL)) {

```

```

        printf("Error en la reserva de espacio para los Vectores o MatrizTri\n");
        exit(-2);
    }
#endif

    srand(N); // esta es la semilla que se usa para los random
    // Inicializamos la Matrices
    for(f = 0; f < N; f++){
        for(c = 0; c < N; c++){
            M1[f][c]=rand()%10;
            M2[f][c]=rand()%10;
        }
    }

    // imprimimos la matriz y el vector si el tamaño de N < PRINT_ALL_MIN
    if (N <= PRINT_ALL_MIN && DEBUGMODE!=1){
        printf("\nEsta es la matriz 1: \n");
        for (f = 0; f < N; f++){
            for (c = 0; c < N; c++){
                printf ("%d ",M1[f][c]);
            }
            printf ("\n");
        }
        printf ("\nEsta es la matriz 2: \n");
        for (f = 0; f < N; f++){
            for (c = 0; c < N; c++){
                printf ("%d ",M2[f][c]);
            }
            printf ("\n");
        }
        printf ("\n");
    }

    // Calcular la multiplicación de una matriz por un vector
    clock_gettime(CLOCK_REALTIME, &t1);
    mejoras(M1,M2,MR,N);
    clock_gettime(CLOCK_REALTIME, &t2);

    // calculamos el tiempo que hemos tardado en calcular la multiplicación
    tr = (double) (t2.tv_sec - t1.tv_sec) + (double) ((t2.tv_nsec - t1.tv_nsec) /
(1.e+9));

    // Ahora imprimimos por pantalla los resultados obtenidos segun las
restricciones del problema
    if (N <= PRINT_ALL_MIN && DEBUGMODE == 0 && boolImprime == 0){
        printf("Tiempo(seg.):%11.9f\nTamaño Matriz y Vector:%u\n",tr,N); // si
queremos imprimir datos completos y N < PRINT_ALL_MIN
        printf ("Este es la matriz resultante: \n");
        for (f = 0; f < N; f++){
            for (c = 0; c < N; c++){
                printf ("%d ",MR[f][c]);
            }
            printf ("\n");
        }
        printf("\n");
    }else if (DEBUGMODE == 1 || boolImprime == 1) // si queremos imprimir unicamente
el tiempo de cálculo
        printf("%11.9f\n",tr); //
    else{ // y si queremos imprimir el tiempo la primera y la ultima multiplicación
        printf("Tiempo(seg.):%11.9f\n",tr);
        printf("Tamaño Matriz 1, Matriz 2 y Matriz resultante: %u\n",N);
        printf("(M1[0][0]=%d)*(M2[0][0]=%d)=%d\n",M1[0][0],M2[0][0],MR[0][0]);
        printf("(M1[%d][%d]=%d)*(M2[%d][%d]=%d)=%d\n",N-1,N-1,M1[N-1][N-1],N-1,N-
1,M2[N-1][N-1],MR[N-1][N-1]);
    }

    for(f=0; f<N; f++){
        free(M1[f]);
        free(M2[f]);
        free(MR[f]);
    }
    free(M1);
    free(M2);
    free(MR);
    return 0;
}

```

Esta parte del código se encuentra en otro fichero totalmente aparte

```

/*
 * pmm-mejoras.c
 *
 * Created on: 25/05/2014
 * Author: Carlos de la Torre
 */
#ifdef MEJORA1
void mejoras(int **M1, int **M2, int **MR, int N){
    int f, c, k;
    int tmpA0=0, tmpA1=0, tmpA2=0, tmpA3=0;
    for (f = 0; f < N; ++f) {
        for (c = 0; c < N; ++c) {
            tmpA0=0; tmpA1=0; tmpA2=0; tmpA3=0;
            for (k = 0; k < N; k+=4) {
                tmpA0 += M1[f][k] * M2[k][c];
                tmpA1 += M1[f][k+1] * M2[k+1][c];
                tmpA2 += M1[f][k+2] * M2[k+2][c];
                tmpA3 += M1[f][k+3] * M2[k+3][c];
            }
            MR[f][c] = tmpA0+tmpA1+tmpA2+tmpA3;
        }
    }
}
#elif defined MEJORA2
void mejoras(int **M1, int **M2, int **MR, int N){
    int f, c, k;
    int tmpA0=0, tmpA1=0, tmpA2=0, tmpA3=0;
    int tmpB0=0, tmpB1=0, tmpB2=0, tmpB3=0;
    for (f = 0; f < N; ++f) {
        for (c = 0; c < N; c+=2) {
            tmpA0=0; tmpA1=0; tmpA2=0; tmpA3=0;
            for (k = 0; k < N; k+=4) {
                tmpA0 += M1[f][k] * M2[k][c];
                tmpA1 += M1[f][k+1] * M2[k+1][c];
                tmpA2 += M1[f][k+2] * M2[k+2][c];
                tmpA3 += M1[f][k+3] * M2[k+3][c];
            }
            MR[f][c] = tmpA0+tmpA1+tmpA2+tmpA3;
            tmpB0=0; tmpB1=0; tmpB2=0; tmpB3=0;
            for (k = 0; k < N; k+=4) {
                tmpB0 += M1[f][k] * M2[k][c+1];
                tmpB1 += M1[f][k+1] * M2[k+1][c+1];
                tmpB2 += M1[f][k+2] * M2[k+2][c+1];
                tmpB3 += M1[f][k+3] * M2[k+3][c+1];
            }
            MR[f][c+1] = tmpB0+tmpB1+tmpB2+tmpB3;
        }
    }
}
#else
void mejoras(int **M1, int **M2, int **MR, int N){
    int f, c, k;
    for (f = 0; f < N; ++f) {
        for (c = 0; c < N; ++c) {
            for (k = 0; k < N; ++k) {
                MR[f][c] += M1[f][k] * M2[k][c];
            }
        }
    }
}
#endif

```

MODIFICACIONES REALIZADAS:

Modificación a) –explicación–:

La primera modificación que se ha realizado ha sido desenrollar el bucle mas interno de la multiplicación de matrices de tal manera que cada iteración se hicieran 4 operaciones.

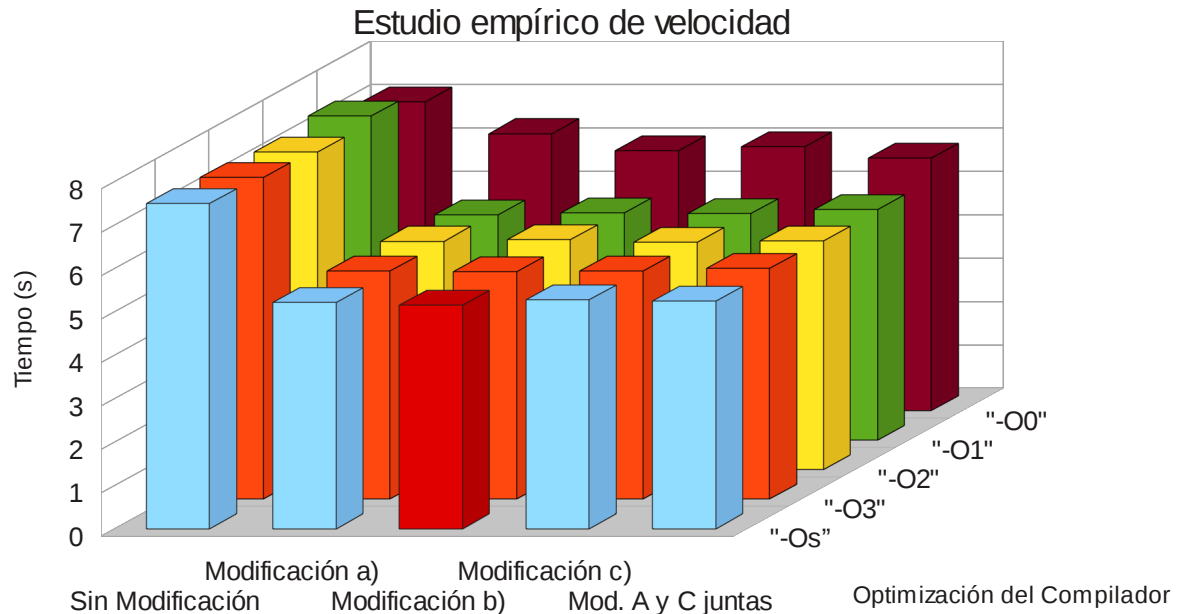
Modificación b) –explicación–:

En la segunda mejora se ha desenrollado el bucle intermedio de la multiplicación pero esta vez se ha dejado en dos operaciones por iteración.

Modificación c) -explicación:-

En la ultima modificación se ha alineado las diferentes matrices en memoria cache.

| PMM | "-O0" | "-O1" | "-O2" | "-O3" | "-Os" |
|-------------------|-------------|-------------|-------------|-------------|-------------|
| Sin Modificación | 7,10992959 | 7,47286821 | 7,317749786 | 7,418983843 | 7,490070101 |
| Modificación a) | 6,377128133 | 5,180591797 | 5,259947191 | 5,2503312 | 5,216687053 |
| Modificación b) | 5,980298209 | 5,236441182 | 5,297300943 | 5,237094457 | 5,149572354 |
| Modificación c) | 6,082424709 | 5,211812884 | 5,232251894 | 5,261426637 | 5,270094339 |
| Mod. A y C juntas | 5,812109558 | 5,305098407 | 5,2626529 | 5,321809524 | 5,248705119 |

**COMENTARIOS SOBRE LOS RESULTADOS:**

La verdad es totalmente sorprendente los resultados que se pueden conseguir solamente con un par de modificaciones del código de alto nivel, esta claro que cuando estemos compilando programas finales tendremos que usar siempre la opción -O2 como mínimo pues la -O3 según en que circunstancias puede elevar el tiempo de ejecución, y también llama la atención, el gran salto en velocidad que se produce solo aplicando el operador -O1, aparte de todo si utilizamos estas técnicas de programación para hacer mas efectivos nuestros códigos conseguiremos CASI los mismos resultados que si aplicáramos este ultimo operador.

Para poder realizar las capturas de pantalla lo que he hecho es un script que me permita ejecutar todos los binarios que se encuentran en el directorio /bin y si le paso el modificado 1 imprime por pantalla solamente los tiempos de ejecución.

Si nos fijamos en las tablas de resultados y en las gráficas podemos apreciar en color rojo cual es el mejor resultado para la multiplicación de matrices.

CAPTURAS DE PANTALLA:

```
--Tiempos de Producto de Matriz por Matriz--
Tiempo(seg.):6.449858601
Tamaño Matriz 1, Matriz 2 y Matriz resultante: 800
(M1[0][0]=3)*(M2[0][0]=2)=15373
(M1[799][799]=8)*(M2[799][799]=1)=16475
Tiempo(seg.):5.206647352
Tamaño Matriz 1, Matriz 2 y Matriz resultante: 800
(M1[0][0]=3)*(M2[0][0]=2)=15373
(M1[799][799]=8)*(M2[799][799]=1)=16475
Tiempo(seg.):5.237254275
Tamaño Matriz 1, Matriz 2 y Matriz resultante: 800
(M1[0][0]=3)*(M2[0][0]=2)=15373
(M1[799][799]=8)*(M2[799][799]=1)=16475
Tiempo(seg.):5.287136936
Tamaño Matriz 1, Matriz 2 y Matriz resultante: 800
(M1[0][0]=3)*(M2[0][0]=2)=15373
(M1[799][799]=8)*(M2[799][799]=1)=16475
Tiempo(seg.):5.236343582
Tamaño Matriz 1, Matriz 2 y Matriz resultante: 800
(M1[0][0]=3)*(M2[0][0]=2)=15373
(M1[799][799]=8)*(M2[799][799]=1)=16475
Tiempo(seg.):6.398892520
Tamaño Matriz 1, Matriz 2 y Matriz resultante: 800
(M1[0][0]=3)*(M2[0][0]=2)=15373
(M1[799][799]=8)*(M2[799][799]=1)=16475
Tiempo(seg.):5.226332306
Tamaño Matriz 1, Matriz 2 y Matriz resultante: 800
(M1[0][0]=3)*(M2[0][0]=2)=15373
(M1[799][799]=8)*(M2[799][799]=1)=16475
Tiempo(seg.):5.262984312
Tamaño Matriz 1, Matriz 2 y Matriz resultante: 800
(M1[0][0]=3)*(M2[0][0]=2)=15373
(M1[799][799]=8)*(M2[799][799]=1)=16475
```

B) CÓDIGO FIGURA 1:**CÓDIGO FUENTE:** figura1-modificado.c

```
/*
 * nucleo.c
 *
 * Created on: 25/05/2014
 * Author: Carlos de la Torre
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define FOREXT 40000
#define FORINT 5000

void mejoras(int *datos, int N_ext, int N_int);

int main(int argc, char* argv[]) {
    /* Modificaciones para ganar tiempo en la ejecución
     * -----
     * Lo primero que hemos hecho es quitar uno de los bucles internos
     * después he hecho un desenrollado de bucles
     * después he alineado el vector donde se guardan los
     */
```

```

    int j;
    double tr;
    struct timespec t1, t2;

#ifdef MEJORA_MEM
    const int LINEA_CACHE = 64;
    const int TAM_BITS = 8192;
    // dos temporales para poder alinear la memoria
    // *tempA, *tempB;
    // el 63 es el valor del tamaño de las líneas de cache del procesador
    int *MEM = (int*) malloc(sizeof(int) * FOREXT + LINEA_CACHE-1);
    // los 63 es tamaño línea de cache
    int *tempA = (int *)(((int)MEM+LINEA_CACHE-1)&~(LINEA_CACHE-1));
    // los 8192 son 32KB de tamaño cache nivel 1
    // por 2 memorias que tiene mi procesador
    // dividido entre 8 vias se quedan 8192
    int *tempB = (int *)((((int)MEM+LINEA_CACHE-1)&~(LINEA_CACHE-1))
+TAM_BITS+LINEA_CACHE);
    if ((MEM == NULL) || (tempA == NULL) || (tempB == NULL)) {
        printf("Error en la reserva de espacio para los Vector o Matriz\n");
        exit(-2);
    }
#else
    /* aquí se hace la reserva de memoria normal con
    * malloc y se devuelve como puntero
    */
    int *MEM = (int*) malloc(sizeof(int) * FOREXT);
    if (MEM == NULL) {
        printf("Error en la reserva de espacio para los Vector o Matriz\n");
        exit(-2);
    }
#endif

// pongo a 0 todo el vector esto solo se usa con malloc
// la funcion calloc es lo mismo que malloc pero llena
// la memoria reservada con 0s
for (j=0;j<FOREXT;++j)
    MEM[j] = 0;

    clock_gettime(CLOCK_REALTIME, &t1);
    mejoras(MEM,FOREXT,FORINT); // esta función se encuentra en los diferentes
    ficheros de mejoras
    clock_gettime(CLOCK_REALTIME, &t2);
    // calculamos el tiempo que hemos tardado en calcular la multiplicación
    tr = (double) (t2.tv_sec - t1.tv_sec) + (double) ((t2.tv_nsec - t1.tv_nsec) /
(1.e+9));
    if (argc == 2 && atoi(argv[1])==1)
        printf("%11.9f\n",tr);
    else
        printf("Tiempo(seg.):%11.9f\n",tr);

    free(MEM);
    return 0;
}
Esta parte del código se encuentra en otro fichero totalmente aparte
/*
 * nucleo_mejora_1_C.c
 *
 * Created on: 25/05/2014
 * Author: Carlos de la Torre
 */
struct {
    int a;
    int b;
} s[5000];

#ifdef MEJORA1
void mejoras(int *datos, int N_ext, int N_int) {
    /* Modificaciones para ganar tiempo en la ejecución
    * -----
    * En esta primera mejora quitamos el segundo for interno
    * que se encarga de realizar las operaciones
    */

```



```

    int ii,i,X1=0,X2=0;

    // pongo a 0 todo el vector
    for (i=0;i<N_ext;++i)
        datos[i] = 0;

    for (ii = 1; ii <= N_ext; ++ii) {
        for (i = 0; i < N_int; ++i){
            X1 = 2 * s[i].a + ii;
            X2 = 3 * s[i].b - ii;
        }

        if (X1<X2)
            datos[ii] = X1;
        else
            datos[ii] = X2;
    }
}
#elif defined MEJORA2
void mejoras(int *datos, int N_ext, int N_int){
    /* Modificaciones para ganar tiempo en la ejecución
    * -----
    * En esta segunda optimización lo que hemos hecho ha
    * sido desenrollar el único bucle interno que quedaba
    * hemos dividido la cantidad de iteraciones en 4
    * y hemos puesto mas líneas de asignación
    */
    int ii,i,X1=0,X2=0;

    int n = N_int/4;
    for (ii = 1; ii <= N_ext; ++ii) {
        for (i = 0; i < n; i+=4){
            X1 = 2 * s[i].a + ii;
            X2 = 3 * s[i].b - ii;
            X1 += 2 * s[i+1].a + ii;
            X2 += 3 * s[i+1].b - ii;
            X1 += 2 * s[i+2].a + ii;
            X2 += 3 * s[i+2].b - ii;
            X1 += 2 * s[i+3].a + ii;
            X2 += 3 * s[i+3].b - ii;
        }

        if (X1<X2)
            datos[ii] = X1;
        else
            datos[ii] = X2;
    }
}
#elif defined MEJORA3
void mejoras(int *datos, int N_ext, int N_int){
    /* Modificaciones para ganar tiempo en la ejecución
    * -----
    * En esta tercera optimización lo que he hecho
    * es quitar las multiplicaciones por 2 y
    * sustituirlas por desplazamientos de bits
    */
    int ii,i,X1=0,X2=0;

    int n = N_int/4;
    for (ii = 1; ii <= N_ext; ++ii) {
        for (i = 0; i < n; i+=4){
            X1 = s[i].a + ii;
            X1 = X1 << 1;
            X2 = 3 * s[i].b - ii;
            X1 += s[i+1].a + ii;
            X1 = X1 << 1;
            X2 += 3 * s[i+1].b - ii;
            X1 += s[i+2].a + ii;
            X1 = X1 << 1;
            X2 += 3 * s[i+2].b - ii;
            X1 += s[i+3].a + ii;
            X1 = X1 << 1;
            X2 += 3 * s[i+3].b - ii;
        }
    }
}

```

```

        if (X1<X2)
            datos[ii] = X1;
        else
            datos[ii] = X2;
    }
}
#else
void mejoras(int *datos, int N_ext, int N_int) {
    /* Modificaciones para ganar tiempo en la ejecución
     * -----
     * En esta opcion no hemos mejorado el codigo
     */
    int ii,i,X1=0,X2=0;

    for (ii = 1; ii <= N_ext; ii++) {
        for (i = 0; i < N_int; i++)
            X1 = 2 * s[i].a + ii;
        for (i = 0; i < N_int; i++)
            X2 = 3 * s[i].b - ii;

        if (X1<X2)
            datos[ii] = X1;
        else
            datos[ii] = X2;
    }
}
#endif

```

MODIFICACIONES REALIZADAS:**Modificación a) -explicación-:**

En esta primera mejora quitamos el segundo for interno que se encarga de realizar las operaciones de multiplicación por 3 y la resta de ii

Modificación b) -explicación-:

En esta segunda optimización lo que hemos hecho ha sido desenrollar el único bucle interno que quedaba hemos dividido la cantidad de iteraciones en 4 y hemos puesto mas líneas de asignación

Modificación c) -explicación-:

En esta tercera optimización lo que he hecho es quitar las multiplicaciones por 2 y sustituirlas por desplazamientos de bits.

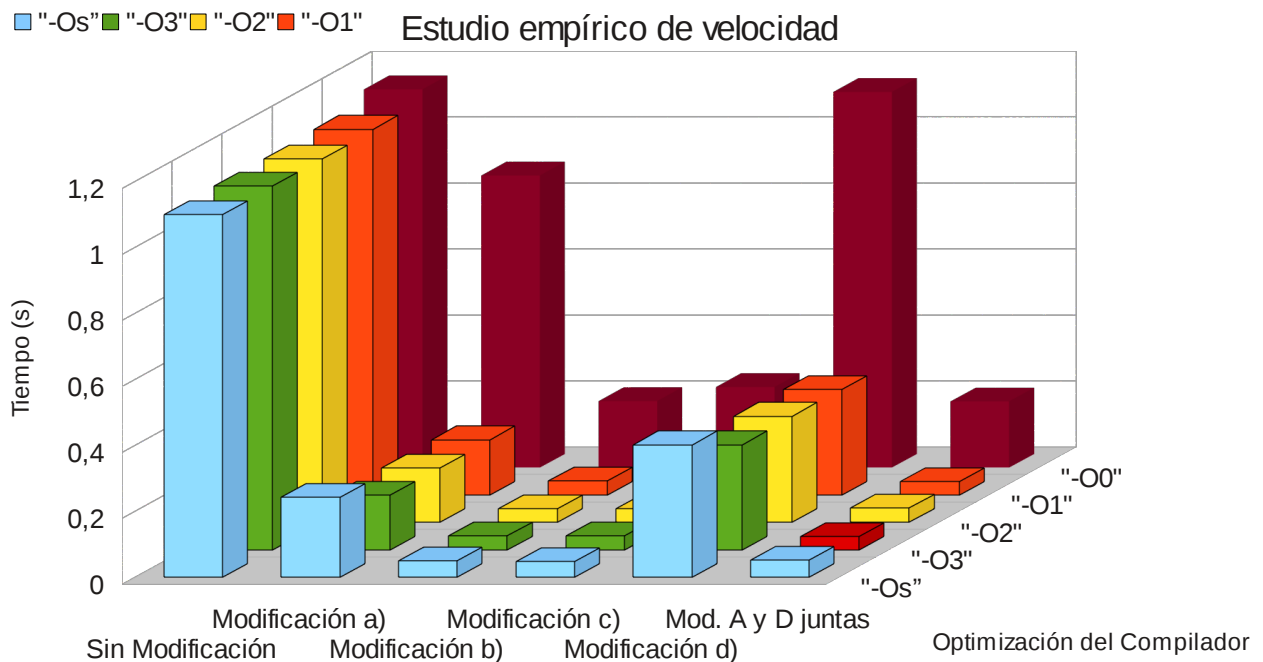
Modificación d) -explicación-:

En la ultima modificación se ha alineado el vector R en memoria cache.

| FIGURA1 | "-O0" | "-O1" | "-O2" | "-O3" | "-Os" |
|-------------------|-------------|-------------|-------------|-------------|-------------|
| Sin Modificación | 1,146654916 | 1,108824838 | 1,101633209 | 1,103570799 | 1,099414352 |
| Modificación a) | 0,885387027 | 0,167181635 | 0,164505207 | 0,166380774 | 0,242934902 |
| Modificación b) | 0,201806127 | 0,042055867 | 0,041562267 | 0,042096466 | 0,049677362 |
| Modificación c) | 0,244082858 | 0,041672078 | 0,04118251 | 0,042105392 | 0,048210089 |
| Modificación d) | 1,138687469 | 0,320039777 | 0,32170802 | 0,31887231 | 0,402027887 |
| Mod. B y D juntas | 0,200519122 | 0,041387666 | 0,044444383 | 0,040962696 | 0,05263849 |

COMENTARIOS SOBRE LOS RESULTADOS:

Como se puede ver tanto en la tabla anterior (en rojo) como en la gráfica posterior, cuando juntamos las mejoras de desenrollado de bucles y alineamiento de memoria para el array R junto con las optimizaciones del compilador en -O3 es donde mejor ganancia de tiempo obtenemos.



CAPTURAS DE PANTALLA:

```

-----Tiempos de Figural-----
figural-mejora1-00
Tiempo(seg.):0.889511964
figural-mejora1-01
Tiempo(seg.):0.165327882
figural-mejora1-02
Tiempo(seg.):0.161488103
figural-mejora1-03
Tiempo(seg.):0.163199121
figural-mejora1-0s
Tiempo(seg.):0.249426349
figural-mejora2-00
Tiempo(seg.):0.203841865
figural-mejora2-01
Tiempo(seg.):0.042382422
figural-mejora2-02
Tiempo(seg.):0.044682343
figural-mejora2-03
Tiempo(seg.):0.042234667
figural-mejora2-0s
Tiempo(seg.):0.052294955
figural-mejora3-00
Tiempo(seg.):0.244175422
figural-mejora3-01
Tiempo(seg.):0.048463201
figural-mejora3-02
Tiempo(seg.):0.042259652
figural-mejora3-03
Tiempo(seg.):0.045197122
figural-mejora3-0s
Tiempo(seg.):0.053645535
figural-mejora_mem-00
Tiempo(seg.):1.123415282
figural-mejora_mem-01
Tiempo(seg.):0.330939256
    
```

2. 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 denominada DAXPY (*Double precision- real Alpha X Plus Y*) que multiplica un vector por una constante y los suma a otro vector (Lección 3/Tema 1):

```
for (i=1;i<=N,i++) y[i]= a*x[i] + y[i];
```

- Genere los programas en ensamblador para cada una de las opciones de optimización del compilador (-O1, -O2,...) y explique las diferencias que se observan en el código justificando las mejoras en velocidad que acarreen.
- (Ejercicio EXTRA) Para la mejor de las opciones, obtenga los tiempos de ejecución con distintos valores de N y determine para su sistema los valores de Rmax (valor máximo del número de operaciones en coma flotante por unidad de tiempo), Nmax (valor de N para el que se consigue Rmax), y N1/2 (valor de N para el que se obtiene Rmax/2). Estime el valor de la velocidad pico (Rpico) del procesador (consulte en [4] el número de ciclos por instrucción punto flotante) y compárela con el valor obtenido para Rmax.

CÓDIGO FUENTE: daxpy.c

```
/*
 * daxpy.c
 *
 * Created on: 25/05/2014
 * Author: Carlos de la Torre
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// #define RAND_MAX 32768
#define DEBUGMODE 0 // <--- Esto es para que se imprima el valor de todos los elementos del vector

int main(int argc, char* argv[]) {
    int i, N, boolImprime = 0;
    double tr, a;
    struct timespec t1, t2;
    switch (argc) { // coneste switch nos aseguramos de que la entrada de parametros sea correcta
        case 1:
            printf("Introduzca el valor de N y del Factor\n");
            printf("\nUso: %s [N] [Factor]\n", argv[0]);
            printf("\nDonde N es el numero de iteraciones del programa\n");
            printf("Es el numero por el que se multiplicara\n");
            exit(-1);
            break;
        case 3:
            N = atoi(argv[1]); // Estas son la cantidad de Iteraciones
            a = atoi(argv[2]); // Este es el valor del Factor
            break;
        case 4:
            N = atoi(argv[1]); // Estas son la cantidad de Iteraciones
            a = atoi(argv[2]); // Este es el valor del Factor
            if (atoi(argv[3]) == 1)
                boolImprime = 1;
            break;
        default:
            printf("La cantidad de parametros es incorrecta\n");
            exit(-1);
            break;
    }
}
```

```

    }
    if (N > 0 && a != 0.0){
        double *x = (int*) malloc(sizeof(int) * N);
        double *y = (int*) malloc(sizeof(int) * N);
        if ((y == NULL) || (x == NULL)) {
            printf("Error en la reserva de espacio para los Vector o
Matriz\n");
            exit(-2);
        }
        srand(time(NULL)); // esta es la semilla que se usa para los random
        // Inicializamos los arrays
        for (i=0;i<N-3;i+=4){
            x[i] = ((double)rand())/((double)RAND_MAX);
            y[i] = ((double)rand())/((double)RAND_MAX);
            x[i+1] = ((double)rand())/((double)RAND_MAX);
            y[i+1] = ((double)rand())/((double)RAND_MAX);
            x[i+2] = ((double)rand())/((double)RAND_MAX);
            y[i+2] = ((double)rand())/((double)RAND_MAX);
            x[i+3] = ((double)rand())/((double)RAND_MAX);
            y[i+3] = ((double)rand())/((double)RAND_MAX);
        }

        clock_gettime(CLOCK_REALTIME, &t1);
        for (i=1;i<=N;i++){
            y[i] = a*x[i] + y[i];
            clock_gettime(CLOCK_REALTIME, &t2);
            tr = (double) (t2.tv_sec - t1.tv_sec) + (double) ((t2.tv_nsec -
t1.tv_nsec) / (1.e+9));
            if (DEBUGMODE==1)
                for (i=1;i<=N;i++)
                    printf("Valor y[i]=%11.9f\n",y[i]);
            if (argc == 4 && boolImprime == 1)
                printf("%11.9f\n",tr);
            else{
                printf("Tiempo(seg.):%11.9f\n",tr);
            }
        }else
            printf("N tiene que ser mayor que 0 y alpha debe de ser diferente de
0.0\n");
        return 0;
    }
}

```

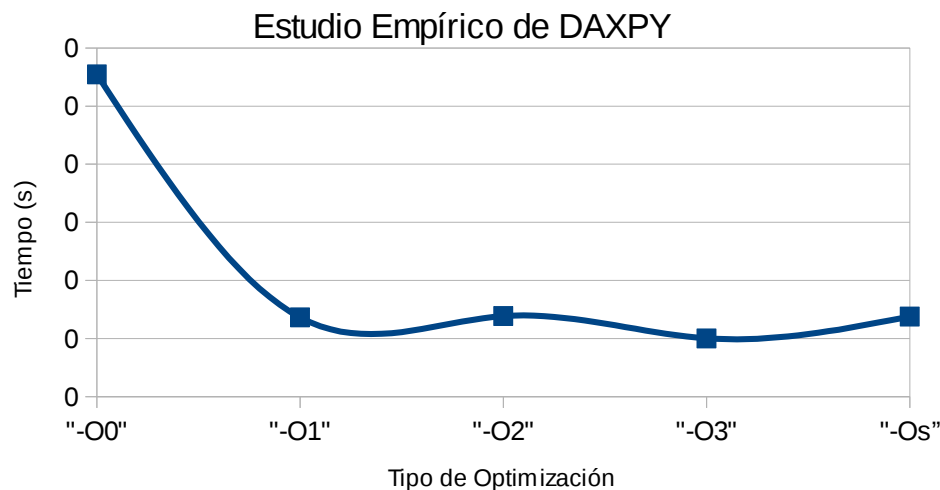
| | "-O0" | "-O1" | "-O2" | "-O3" | "-Os" |
|-------|-------------|-------------|-------------|-------------|-------------|
| DAXPY | 0,000110944 | 0,000027238 | 0,000027691 | 0,000020064 | 0,000027561 |

CAPTURAS DE PANTALLA:

```

-----Tiempos de DAXPY-----
daxpy-00
Tiempo(seg.):0.000120864
daxpy-01
Tiempo(seg.):0.000029639
daxpy-02
Tiempo(seg.):0.000027567
daxpy-03
Tiempo(seg.):0.000015739
daxpy-0s
Tiempo(seg.):0.000025290

```



COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:

Lo primero que hay que decir es como se puede encontrar las líneas correspondientes al bucle y la sentencia de multiplicación suma y asignación del programa, en mi caso como se tenía que usar la llamada a la función `clock_gettime()`, estaba claro que el código que hubiera entre estas dos llamadas sería el código del núcleo del programa DAXPY, una vez localizadas todas las líneas en las diferentes versiones que genera el compilador, lo único que queda es compararlas de tal forma que primero comparare la versión -O0 con la versión -O1:

Lo primero que llama la atención en esta optimización es la clara reducción de líneas de código ensamblador ya que en la primera tiene 29 líneas y en la segunda solo 13 líneas, esto claramente se traduce en un aumento de velocidad inmediato, aparte de esto en el segundo caso se puede apreciar como se utilizan menos registros para hacer las operaciones.

Si comparamos -O1 con -O2 nos damos cuenta que la reducción de líneas de código no es tan evidente como en el caso anterior pero sin embargo si nos damos cuenta de que en esta ocasión hay solo una sección que se encarga de tanto de la multiplicación suma y asignación como del bucle, cosa que antes se hacía en dos secciones diferentes.

Comparando -O2 con -O3 podemos ver que el código crece considerablemente, y aunque no estoy seguro de esto por que no he realizado un análisis a fondo, creo que es debido a que divide las tareas a realizar en las diferentes secciones que se encuentran entre las dos llamadas de `clock_gettime()`. Y por último si comparamos la versión -O3 con la versión -Os vemos que aunque la segunda tiene muchísimas menos líneas de código la versión -O3, esta última es la que mejor rendimiento obtiene, esto puede ser debido a que el compilador puede realizar un desenrollado de bucle y por eso tiene más líneas pero es más eficiente, también me he dado cuenta que esta última versión -Os se diferencia en dos líneas de la versión -O2

CÓDIGO EN ENSAMBLADOR: (LIMITAR AQUÍ EL CÓDIGO INCLUIDO A LA ZONA DEL CÓDIGO ENSAMBLADOR DONDE SE REALIZA LA OPERACIÓN CON VECTORES)

daxpy00.s

```
.L15:
    call    clock_gettime
    movl    $1, -20(%rbp)
    jmp     .L14
    movl    -20(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -56(%rbp), %rax
    addq    %rdx, %rax
    movl    -20(%rbp), %edx
```

```

movslq %edx, %rdx
leaq 0(,%rdx,8), %rcx
movq -48(%rbp), %rdx
addq %rcx, %rdx
movsd (%rdx), %xmm0
mulsd -40(%rbp), %xmm0
movl -20(%rbp), %edx
movslq %edx, %rdx
leaq 0(,%rdx,8), %rcx
movq -56(%rbp), %rdx
addq %rcx, %rdx
movsd (%rdx), %xmm1
addsd %xmm1, %xmm0
movsd %xmm0, (%rax)
addl $1, -20(%rbp)
.L14:
movl -20(%rbp), %eax
cmpl -24(%rbp), %eax
jle .L15
leaq -96(%rbp), %rax
movq %rax, %rsi
movl $0, %edi
call clock_gettime

```

daxpy01.s

```

call clock_gettime
cmpb $0, 27(%rsp)
je .L12
movl $1, %eax
.L13:
movslq %eax, %rcx
leaq (%r12,%rcx,8), %rdx
movsd 8(%rsp), %xmm0
mulsd 0(%rbp,%rcx,8), %xmm0
addsd (%rdx), %xmm0
movsd %xmm0, (%rdx)
addl $1, %eax
cmpl %eax, 4(%rsp)
jge .L13
.L12:
leaq 32(%rsp), %rsi
movl $0, %edi
call clock_gettime

```

daxpy02.s

```

call clock_gettime
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L11:
movsd (%rsp), %xmm0
mulsd 8(%r12,%rax,8), %xmm0
addsd 8(%rbp,%rax,8), %xmm0
movsd %xmm0, 8(%rbp,%rax,8)
addq $1, %rax
leal 1(%rax), %edx
cmpl %edx, %r13d
jge .L11
leaq 32(%rsp), %rsi
xorl %edi, %edi
call clock_gettime

```

daxpy03.s

```

call clock_gettime
leaq 8(%rbp), %rax
testl %r15d, %r15d
movl $1, %edx
cmovg %r15d, %edx
salq $60, %rax
movl %edx, %ecx
shrq $63, %rax

```

```

        cml    %edx, %eax
        cmova  %edx, %eax
        cml    $3, %edx
        ja     .L47
.L11:    xorl    %eax, %eax
.L14:    movsd   (%rsp), %xmm0
        leal    2(%rax), %r9d
        mulsd   8(%r12,%rax,8), %xmm0
        addsd   8(%rbp,%rax,8), %xmm0
        movsd   %xmm0, 8(%rbp,%rax,8)
        addq    $1, %rax
        cml    %eax, %ecx
        ja     .L14
        cml    %edx, %ecx
        je     .L17
.L12:    subl    %ecx, %edx
        movl    %ecx, %eax
        movl    %edx, %r8d
        shr     %r8d
        movl    %r8d, %r10d
        addl    %r10d, %r10d
        je     .L18
        movsd   (%rsp), %xmm1
        leaq    8(,%rax,8), %rsi
        xorl    %ecx, %ecx
        xorl    %eax, %eax
        unpcklpd %xmm1, %xmm1
        leaq    (%r12,%rsi), %rdi
        addq    %rbp, %rsi
.L21:    movsd   (%rdi,%rax), %xmm0
        addl    $1, %ecx
        movhpd  8(%rdi,%rax), %xmm0
        mulpd   %xmm1, %xmm0
        addpd   (%rsi,%rax), %xmm0
        movapd  %xmm0, (%rsi,%rax)
        addq    $16, %rax
        cml    %ecx, %r8d
        ja     .L21
        addl    %r10d, %r9d
        cml    %r10d, %edx
        je     .L17
.L18:    movsd   (%rsp), %xmm0
        movslq   %r9d, %r9
        leaq    0(%rbp,%r9,8), %rax
        mulsd   (%r12,%r9,8), %xmm0
        addsd   (%rax), %xmm0
        movsd   %xmm0, (%rax)
.L17:    leaq    32(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime

```

daxpy0s.s

```

call    clock_gettime
.L13:    xorl    %eax, %eax
        movsd   (%rsp), %xmm0
        mulsd   8(%r12,%rax,8), %xmm0
        addsd   8(%rbp,%rax,8), %xmm0
        movsd   %xmm0, 8(%rbp,%rax,8)
        incq    %rax
        leal    1(%rax), %edx
        cml    %r13d, %edx
        jle     .L13
        leaq    32(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime

```