

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): Rodrigo Raya Castellano

Grupo de prácticas: Viernes

Fecha de entrega:

Fecha evaluación en clase:

**Versión de gcc utilizada:** comando gcc -v, gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

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

1. Para el núcleo que se muestra en la Figura 1 (ver guion de prácticas), 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,...) e incorpórelos al cuaderno de prácticas. 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. Destaque las diferencias en el código ensamblador.
  - 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

**A) MULTIPLICACIÓN DE MATRICES:****CÓDIGO FUENTE: prod0.c (versión original)**

```

#include <stdio.h>
#include <time.h>
//producto A=B*C
unsigned int N = 1000;
//If the array is declared as a global one or as static in a function,
//then all elements are initialized to zero if they aren't initialized
already.
double A[1000][1000],B[1000][1000],C[1000][1000];
void main() {
    int i,j,k;
    struct timespec cgt1,cgt2;
    double ncgt;
    clock_gettime(CLOCK_REALTIME,&cgt1);
    double tmp[N][N];
    for (i=0;i<N;++i)//trasponemos C
        for (j = 0; j < N; ++j)
            tmp[i][j] = C[j][i];
    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            for (k = 0; k < N; ++k)
                A[i][j] += B[i][k] * tmp[j][k];
    clock_gettime(CLOCK_REALTIME,&cgt2);
    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double) ((cgt2.tv_nsec-
cgt1.tv_nsec)/(1.e+9));
    //Imprimir resultados
    printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
}

```

**CÓDIGO FUENTE: prod1.c (versión 1)**

```

#include <stdio.h>
#include <time.h>
//producto A=B*C
unsigned int N = 1000;
//If the array is declared as a global one or as static in a function,
//then all elements are initialized to zero if they aren't initialized
already.
int A[1000][1000],B[1000][1000],C[1000][1000];
void main() {
    int i,j,k;
    struct timespec cgt1,cgt2;
    double ncgt;
    clock_gettime(CLOCK_REALTIME,&cgt1);
    int tmp[N][N];
    for (i=0;i<N;++i)//trasponemos C
        for (j = 0; j < N; ++j)
            tmp[i][j] = C[j][i];
    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            for (k = 0; k < N; ++k)
                A[i][j] += B[i][k] * tmp[j][k];
    clock_gettime(CLOCK_REALTIME,&cgt2);
    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double) ((cgt2.tv_nsec-
cgt1.tv_nsec)/(1.e+9));
    //Imprimir resultados
    printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
}

```

**CÓDIGO FUENTE:** prod4.c (versión 2)

```

#include <stdio.h>
#include <time.h>
//producto A=B*C
unsigned int N = 1000;
//If the array is declared as a global one or as static in a function,
//then all elements are initialized to zero if they aren't initialized
already.
int A[1008][1008] __attribute__((aligned (16)));
int B[1008][1008] __attribute__((aligned (16)));
int C[1008][1008] __attribute__((aligned (16)));
void main() {
    int i,j,k;
    struct timespec cgt1,cgt2;
    double ncgt;
    clock_gettime(CLOCK_REALTIME,&cgt1);
    int tmp[N][N];
    for (i=0;i<N;++i)//trasponemos C
        for (j = 0; j < N; ++j)
            tmp[i][j] = C[j][i];

    for (i=0;i<N;i+=16){
        for (j=0;j<N;j++){
            for (k=0;k<N;k++){
                C[i][j]+= A[i][k]*tmp[j][k];
                C[i+1][j]+= A[i+1][k]*tmp[j][k];
                C[i+2][j]+= A[i+2][k]*tmp[j][k];
                C[i+3][j]+= A[i+3][k]*tmp[j][k];
                C[i+4][j]+= A[i+4][k]*tmp[j][k];
                C[i+5][j]+= A[i+5][k]*tmp[j][k];
                C[i+6][j]+= A[i+6][k]*tmp[j][k];
                C[i+7][j]+= A[i+7][k]*tmp[j][k];
                C[i+8][j]+= A[i+8][k]*tmp[j][k];
                C[i+9][j]+= A[i+9][k]*tmp[j][k];
                C[i+10][j]+= A[i+10][k]*tmp[j][k];
                C[i+11][j]+= A[i+11][k]*tmp[j][k];
                C[i+12][j]+= A[i+12][k]*tmp[j][k];
                C[i+13][j]+= A[i+13][k]*tmp[j][k];
                C[i+14][j]+= A[i+14][k]*tmp[j][k];
                C[i+15][j]+= A[i+15][k]*tmp[j][k];
            }
        }
    }
    clock_gettime(CLOCK_REALTIME,&cgt2);
    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double) ((cgt2.tv_nsec-
cgt1.tv_nsec)/(1.e+9));
    //Imprimir resultados
    printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
}

```

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

La modificación a) puede consultarse en prod4.c. Se trata básicamente de un desenrollado de bucle.

**Modificación b) –explicación-:**

La modificación b) puede consultarse en prod1.c y sigue algunas ideas del texto “What Every Programmer Should Know About Memory” de Ulrich Drepper. Se trata de transponer la matriz de la derecha con el fin de que el acceso a datos se corresponda con la forma en que se almacenan las matrices en C.

Las modificaciones posteriores de Drepper son sumamente interesantes. La idea general era evitar fallos de caché haciendo los productos escalares justos que caben en una línea de caché. El estudio continúa hasta una versión muy óptima que está recogida en otra.c.

**Tabla de tiempos**

<b>Modificación</b>	<b>-O0</b>	<b>-O1</b>	<b>-O2</b>	<b>-O3</b>	<b>-Os</b>
Sin modificar	6.910679635	1.684278112	1.680973513	1.664086916	3.073371927
Modificación a)	4.725481771	0.955217105	0.963067915	0.609423356	2.583595652
Modificación b)	4.447982340	0.897989836	0.563496826	0.439934272	0.873181692

**COMENTARIOS SOBRE LOS RESULTADOS:**

Como la modificación b) recoge las ideas de la modificación a) estimamos conveniente comparar tan sólo la versión sin modificar y la versión b).

**Códigos ensamblador****Ensamblador optimización 1 primera versión**

```

call    clock_gettime
        movl    N, %eax
        movl    %eax, 20(%esp)
        testl   %eax, %eax
        je      .L2
        movl    $0, 28(%esp)
        movl    $0, 40(%esp)
        movl    $0, 24(%esp)
        sall    $2, %eax
        movl    %eax, 32(%esp)
        jmp     .L3
.L6:    //Bucle más interno (resto son bucles de control)
        movl    (%eax), %ebx
        imull    (%edx), %ebx
        addl    %ebx, %ecx
        addl    $4, %eax
        addl    $4000, %edx
        cmpl    %esi, %eax
        jne     .L6
        movl    40(%esp), %eax
        movl    %ecx, A(%eax,%edi)
        addl    $4, %edi
        cmpl    32(%esp), %edi
        je      .L5
.L7:    movl    40(%esp), %eax
        movl    A(%eax,%edi), %ecx
        leal    C(%edi), %edx
        movl    36(%esp), %eax
        jmp     .L6
.L5:    addl    $1, 24(%esp)
        addl    $4000, 40(%esp)
        addl    $1000, 28(%esp)
        movl    24(%esp), %eax
        cmpl    %eax, 20(%esp)
        jbe     .L2
.L3:    movl    20(%esp), %edi
        testl   %edi, %edi
        je      .L5
        movl    28(%esp), %eax
        addl    %edi, %eax
        leal    B(,%eax,4), %esi
        movl    $0, %edi
        movl    40(%esp), %eax
        addl    $B, %eax
        movl    %eax, 36(%esp)
        jmp     .L7
.L2:    ...
        call    clock_gettime

```

## Ensamblador optimización 1 última versión

ccall	clock_gettime
	movl N, %eax
	movl %eax, %ebx
	movl %eax, -64(%ebp)
	leal 0(,%eax,4), %esi
	movl %eax, %edi
	imull %eax, %edi
	leal 18(,%edi,4), %eax
	andl \$-16, %eax
	subl %eax, %esp
	leal 20(%esp), %edi
	movl %ebx, %eax
	testl %ebx, %ebx
	je .L2 //termina
	movl %esi, -56(%ebp)
	movl %edi, -76(%ebp)
	leal C(%esi), %ebx
	movl %ebx, -48(%ebp)
	imull \$4032, %eax, %eax
	movl %eax, -52(%ebp)
	movl \$C, %ebx
	jmp .L3
.L6:	
	movl (%eax), %ebx
	movl %ebx, (%edx)
	addl \$4032, %eax
	addl \$4, %edx
	cmpl %ecx, %eax
	jne .L6
	movl -60(%ebp), %ebx
	addl \$4, %ebx
	addl -56(%ebp), %edi
	cmpl -48(%ebp), %ebx
	je .L5
.L3:	
	movl -52(%ebp), %eax
	leal (%ebx,%eax), %ecx
	movl %edi, %edx
	movl %ebx, %eax
	movl %ebx, -60(%ebp)
	jmp .L6
.L5:	
	movl -64(%ebp), %eax
	sall \$2, %eax
	negl %eax
	movl %eax, -72(%ebp)
	movl -48(%ebp), %eax
	movl %eax, -52(%ebp)
	movl \$0, -68(%ebp)
	movl \$A, %eax
	subl \$C, %eax
	movl %eax, -80(%ebp)
	subl %esi, %eax
	movl %eax, -84(%ebp)
	jmp .L7
.L10:	
	movl (%ebx), %ecx
	movl %ecx, %esi
	imull (%edx), %esi
	addl %esi, (%eax)

	movl	%ecx, %esi
	imull	4032(%edx), %esi
	addl	%esi, 4032(%eax)
	movl	%ecx, %esi
	imull	8064(%edx), %esi
	addl	%esi, 8064(%eax)
	imull	12096(%edx), %ecx
	addl	%ecx, 12096(%eax)
	addl	\$4, %edx
	addl	\$4, %ebx
	cmpl	%edi, %edx
	jne	.L10
	movl	-56(%ebp), %ebx
	addl	%ebx, -48(%ebp)
	addl	\$4, %eax
	cmpl	-52(%ebp), %eax
	je	.L9
.L11:	movl	-48(%ebp), %ebx
	movl	-60(%ebp), %edx
	jmp	.L10
.L9:	addl	\$4, -68(%ebp)
	addl	\$16128, -52(%ebp)
	movl	-68(%ebp), %eax
	cmpl	%eax, -64(%ebp)
	jbe	.L2 //termina
.L7:	movl	-84(%ebp), %eax
	movl	-52(%ebp), %edi
	addl	%edi, %eax
	movl	%eax, -60(%ebp)
	movl	-72(%ebp), %ebx
	leal	(%edi,%ebx), %eax
	movl	-76(%ebp), %ebx
	movl	%ebx, -48(%ebp)
	addl	-80(%ebp), %edi
	jmp	.L11
.L2:	leal	-32(%ebp), %eax
	movl	%eax, 4(%esp)
	movl	\$0, (%esp)
	call	clock_gettime

La primera diferencia que se advierte es la extensión de la última versión. El desenrollado de bucle tiene este negativo. Otro detalle interesante es que la versión última no tiene ningún incremento \$1 de los contadores mientras que la versión anterior sí. Directamente se accede con la dirección del array. Probablemente al haber mayor cantidad de código el compilador haya considerado en más ocasiones que este cambio es significativo.

## Ensamblador optimización 2 primera versión

	call	clock_gettime
	movl	N, %eax
	testl	%eax, %eax
	movl	%eax, 24(%esp)
	je	.L2
	sall	\$2, %eax
	movl	%eax, 36(%esp)
	addl	\$B, %eax
	movl	\$0, 40(%esp)
	movl	\$0, 28(%esp)
	movl	%eax, 20(%esp)
.L3:		
	movl	40(%esp), %eax
	xorl	%edi, %edi
	movl	20(%esp), %esi
	addl	%eax, %esi
	addl	\$B, %eax
	movl	%eax, 32(%esp)
	.p2align	4,,7
	.p2align	3
.L7:		
	movl	40(%esp), %eax
	leal	C(%edi), %ecx
	movl	A(%eax,%edi), %ebx
	movl	32(%esp), %eax
	.p2align	4,,7
	.p2align	3
.L6:		
	movl	(%eax), %edx
	addl	\$4, %eax
	addl	\$4000, %ecx
	imull	-4000(%ecx), %edx
	addl	%edx, %ebx
	cmpl	%esi, %eax
	jne	.L6
	movl	40(%esp), %eax
	movl	%ebx, A(%eax,%edi)
	addl	\$4, %edi
	cmpl	36(%esp), %edi
	jne	.L7
	addl	\$1, 28(%esp)
	movl	24(%esp), %eax
	addl	\$4000, 40(%esp)
	cmpl	%eax, 28(%esp)
	jne	.L3
.L2:		
	leal	56(%esp), %eax
	movl	%eax, 4(%esp)
	movl	\$0, (%esp)
	call	clock_gettime



## Ensamblador optimización 2 última versión

```

call    clock_gettime
        movl    N, %eax
        movl    %eax, %esi
        movl    %eax, %edi
        imull   %eax, %esi
        movl    %eax, -72(%ebp)
        leal    0(,%eax,4), %edx
        leal    18(%esi,4), %eax
        andl    $-16, %eax
        subl    %eax, %esp
        testl   %edi, %edi
        leal    20(%esp), %eax
        movl    %eax, %esi
        movl    %eax, -108(%ebp)
        movl    %edi, %eax
        je      .L2
        movl    %edx, %edi
        movl    $C, %ebx
        addl    $C, %edi
        movl    %edi, -100(%ebp)
        imull   $4032, %eax, %edi
        movl    %edx, -92(%ebp)

.L3:
        leal    (%ebx,%edi), %ecx
        movl    %esi, %edx
        movl    %ebx, %eax
        movl    %ebx, -48(%ebp)
        .p2align 4,,7
        .p2align 3

.L6:
        movl    (%eax), %ebx
        addl    $4032, %eax
        addl    $4, %edx
        movl    %ebx, -4(%edx)
        cmpl    %eax, %ecx
        jne     .L6
        movl    -48(%ebp), %ebx
        addl    -92(%ebp), %esi
        addl    $4, %ebx
        cmpl    -100(%ebp), %ebx
        jne     .L3
        movl    $0, -88(%ebp)
        movl    $0, -104(%ebp)

.L7:
        movl    -88(%ebp), %esi
        movl    -100(%ebp), %edi
        movl    $12096, -56(%ebp)
        movl    %esi, %eax
        negl    %eax
        subl    %esi, %edi
        movl    %eax, -76(%ebp)
        movl    %esi, %eax
        movl    -108(%ebp), %esi
        subl    %eax, -56(%ebp)
        movl    %edi, -96(%ebp)
        movl    %esi, -68(%ebp)
        movl    $C, %esi
        subl    %eax, %esi
        movl    %esi, -80(%ebp)
        movl    $4032, %esi

```

```

        subl    %eax, %esi
        movl    %esi, -60(%ebp)
        movl    $8064, %esi
        subl    %eax, %esi
        movl    %esi, -64(%ebp)
        .p2align 4,,7
        .p2align 3
.L11:
        movl    -80(%ebp), %eax
        movl    -64(%ebp), %ebx
        movl    -56(%ebp), %edx
        movl    -60(%ebp), %esi
        movl    (%eax), %edi
        addl    -88(%ebp), %eax
        movl    (%eax,%ebx), %ebx
        movl    (%eax,%edx), %ecx
        movl    %eax, -84(%ebp)
        movl    (%eax,%esi), %esi
        xorl    %eax, %eax
        movl    %ebx, -52(%ebp)
        movl    %ecx, -48(%ebp)
        jmp     .L10
        .p2align 4,,7
        .p2align 3
.L8:
        movl    %ebx, -52(%ebp)
        movl    %edx, -48(%ebp)
.L10:
        //Hay 4 imull bucle más interno
        movl    -76(%ebp), %ebx
        movl    -68(%ebp), %edx
        movl    A(%ebx,%eax,4), %ecx
        movl    (%edx,%eax,4), %edx
        movl    -60(%ebp), %ebx
        imull    %edx, %ecx
        addl    %ecx, %edi
        movl    A(%ebx,%eax,4), %ecx
        movl    -64(%ebp), %ebx
        imull    %edx, %ecx
        addl    %ecx, %esi
        movl    A(%ebx,%eax,4), %ecx
        movl    -52(%ebp), %ebx
        imull    %edx, %ecx
        addl    %ecx, %ebx
        movl    -56(%ebp), %ecx
        imull    A(%ecx,%eax,4), %edx
        addl    $1, %eax
        addl    -48(%ebp), %edx
        cmpl    -72(%ebp), %eax
        jne     .L8
        movl    -80(%ebp), %eax
        movl    %edi, -48(%ebp)
        movl    %esi, %edi
        movl    -48(%ebp), %esi
        addl    $4, -80(%ebp)
        movl    %esi, (%eax)
        movl    -84(%ebp), %eax
        movl    -60(%ebp), %esi
        movl    %edi, (%eax,%esi)
        movl    -64(%ebp), %edi
        movl    %ebx, (%eax,%edi)
        movl    -56(%ebp), %edi
        movl    %edx, (%eax,%edi)

```

	movl	-92(%ebp), %eax
	addl	%eax, -68(%ebp)
	movl	-80(%ebp), %eax
	cmpl	-96(%ebp), %eax
	jne	.L11
	addl	\$4, -104(%ebp)
	movl	-104(%ebp), %eax
	subl	\$16128, -88(%ebp)
	cmpl	%eax, -72(%ebp)
	ja	.L7
.L2:	leal	-32(%ebp), %eax
	movl	%eax, 4(%esp)
	movl	\$0, (%esp)
	call	clock_gettime

En ambas versiones observamos un interés del compilador por alinear código con la directiva `align`. Ahora vemos más claro (ver aclaración en el código) cómo el desenrollado se refleja en el código, a partir de L10 vemos 4 `imull`. Realmente, debe ser costoso hacer a memoria porque los `imull` son costosos del orden de 15 a 18 ciclos en las tablas de intel. Nos interesa localizar también dónde se está trasponiendo la matriz. Esto lo encontramos claramente en el bucle anidado de las etiquetas L3 y L6.

Finalmente, nos interesaría mostrar por qué en esta optimización es ya significativamente mejor una versión que la otra. Una posible respuesta se puede encontrar al analizar los bucles. En la primera versión:

L6:

```

movl (%eax), %edx
addl $4, %eax
addl $4000, %ecx
imull -4000(%ecx), %edx
addl %edx, %ebx
cmpl %esi, %eax
jne .L6

```

y en la segunda:

.L10:

```

movl -76(%ebp), %ebx
movl -68(%ebp), %edx
movl A(%ebx,%eax,4), %ecx
movl (%edx,%eax,4), %edx
movl -60(%ebp), %ebx
imull %edx, %ecx
addl %ecx, %edi
movl A(%ebx,%eax,4), %ecx
movl -64(%ebp), %ebx
imull %edx, %ecx
addl %ecx, %esi
movl A(%ebx,%eax,4), %ecx
movl -52(%ebp), %ebx
imull %edx, %ecx
addl %ecx, %ebx
movl -56(%ebp), %ecx
imull A(%ecx,%eax,4), %edx

```

```

    addl    $1, %eax
    addl    -48(%ebp), %edx
    cmpl    -72(%ebp), %eax
    jne     .L8

```

y vemos que mientras en la primera los operandos de los `imull` son traídos de memoria, en la segunda, la mayoría son traídos de registros y anteriormente de la pila. Esto era esperable.

#### Ensamblador optimización 3 primera versión

```

call    clock_gettime
    movl    N, %eax
    testl   %eax, %eax
    movl    %eax, 24(%esp)
    je      .L2
    sall    $2, %eax
    movl    %eax, 36(%esp)
    addl    $B, %eax
    movl    $0, 40(%esp)
    movl    $0, 28(%esp)
    movl    %eax, 20(%esp)
.L3:
    movl    40(%esp), %eax
    xorl    %edi, %edi
    movl    20(%esp), %esi
    addl    %eax, %esi
    addl    $B, %eax
    movl    %eax, 32(%esp)
    movl    40(%esp), %eax
    .p2align 4,,7
    .p2align 3
.L7:
    movl    A(%eax,%edi), %ebx
    leal    C(%edi), %ecx
    movl    32(%esp), %eax
    .p2align 4,,7
    .p2align 3
.L6:
    movl    (%eax), %edx
    addl    $4, %eax
    addl    $4000, %ecx
    imull   -4000(%ecx), %edx
    addl    %edx, %ebx
    cmpl    %esi, %eax
    jne     .L6
    movl    40(%esp), %eax
    movl    %ebx, A(%eax,%edi)
    addl    $4, %edi
    cmpl    36(%esp), %edi
    jne     .L7
    addl    $1, 28(%esp)
    movl    24(%esp), %eax
    addl    $4000, 40(%esp)
    cmpl    %eax, 28(%esp)
    jne     .L3
.L2:
    leal    56(%esp), %eax
    movl    %eax, 4(%esp)
    movl    $0, (%esp)
    call    clock_gettime

```

## Ensamblador optimización 3 última versión

```

        call    clock_gettime
        movl    N, %eax
        movl    %eax, %esi
        movl    %eax, %edi
        imull   %eax, %esi
        movl    %eax, -72(%ebp)
        leal    0(,%eax,4), %edx
        leal    18(%esi,4), %eax
        andl    $-16, %eax
        subl    %eax, %esp
        testl   %edi, %edi
        leal    20(%esp), %eax
        movl    %eax, %esi
        movl    %eax, -108(%ebp)
        movl    %edi, %eax
        je      .L2
        movl    %edx, %edi
        movl    $C, %ebx
        addl    $C, %edi
        movl    %edi, -100(%ebp)
        imull   $4032, %eax, %edi
        movl    %edx, -92(%ebp)
.L3:
        leal    (%ebx,%edi), %ecx
        movl    %esi, %edx
        movl    %ebx, %eax
        movl    %ebx, -48(%ebp)
        .p2align 4,,7
        .p2align 3
.L6:
        movl    (%eax), %ebx
        addl    $4032, %eax
        addl    $4, %edx
        movl    %ebx, -4(%edx)
        cmpl    %eax, %ecx
        jne     .L6
        movl    -48(%ebp), %ebx
        addl    -92(%ebp), %esi
        addl    $4, %ebx
        cmpl    -100(%ebp), %ebx
        jne     .L3
        movl    $0, -88(%ebp)
        movl    $0, -104(%ebp)
.L7:
        movl    -88(%ebp), %esi
        movl    -100(%ebp), %edi
        movl    $12096, -56(%ebp)
        movl    %esi, %eax
        negl    %eax
        subl    %esi, %edi
        movl    %eax, -76(%ebp)
        movl    %esi, %eax
        movl    -108(%ebp), %esi
        subl    %eax, -56(%ebp)
        movl    %edi, -96(%ebp)
        movl    %esi, -68(%ebp)
        movl    $C, %esi
        subl    %eax, %esi

```

```

        movl    %esi, -80(%ebp)
        movl    $4032, %esi
        subl    %eax, %esi
        movl    %esi, -60(%ebp)
        movl    $8064, %esi
        subl    %eax, %esi
        movl    -80(%ebp), %eax
        movl    %esi, -64(%ebp)
        movl    -60(%ebp), %esi
        .p2align 4,,7
        .p2align 3
.L11:
        movl    -64(%ebp), %ebx
        movl    -56(%ebp), %edx
        movl    (%eax), %edi
        addl    -88(%ebp), %eax
        movl    (%eax,%ebx), %ebx
        movl    (%eax,%edx), %ecx
        movl    %eax, -84(%ebp)
        movl    (%eax,%esi), %esi
        xorl    %eax, %eax
        movl    %ebx, -52(%ebp)
        movl    %ecx, -48(%ebp)
        jmp     .L10
        .p2align 4,,7
        .p2align 3
.L8:
        movl    %ebx, -52(%ebp)
        movl    %edx, -48(%ebp)
.L10:
        movl    -76(%ebp), %ebx
        movl    -68(%ebp), %edx
        movl    A(%ebx,%eax,4), %ecx
        movl    (%edx,%eax,4), %edx
        movl    -60(%ebp), %ebx
        imull   %edx, %ecx
        addl    %ecx, %edi
        movl    A(%ebx,%eax,4), %ecx
        movl    -64(%ebp), %ebx
        imull   %edx, %ecx
        addl    %ecx, %esi
        movl    A(%ebx,%eax,4), %ecx
        movl    -52(%ebp), %ebx
        imull   %edx, %ecx
        addl    %ecx, %ebx
        movl    -56(%ebp), %ecx
        imull   A(%ecx,%eax,4), %edx
        addl    $1, %eax
        addl    -48(%ebp), %edx
        cmpl    -72(%ebp), %eax
        jne     .L8
        movl    -80(%ebp), %eax
        movl    %edi, -48(%ebp)
        movl    %esi, %edi
        movl    -48(%ebp), %esi
        addl    $4, -80(%ebp)
        movl    %esi, (%eax)
        movl    -84(%ebp), %eax
        movl    -60(%ebp), %esi
        movl    %edi, (%eax,%esi)
        movl    -64(%ebp), %edi
        movl    %ebx, (%eax,%edi)

```

```

        movl    -56(%ebp), %edi
        movl    %edx, (%eax,%edi)
        movl    -92(%ebp), %eax
        addl    %eax, -68(%ebp)
        movl    -80(%ebp), %eax
        cmpl    -96(%ebp), %eax
        jne     .L11
        addl    $4, -104(%ebp)
        movl    -104(%ebp), %eax
        subl    $16128, -88(%ebp)
        cmpl    %eax, -72(%ebp)
        ja      .L7
.L2:
        leal    -32(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $0, (%esp)
        call    clock_gettime

```

Sin embargo, en esta ocasión no podemos decir gran cosa, lo esencial del ensamblador se preserva en la versión nueva aunque eventualmente desaparecen algunos mov. Creo esto forma parte de las profundidades del compilador y no sé si ganaríamos algo entrando en más detalle.

### CAPTURAS DE PANTALLA:

```

rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$
Tiempo(seg.):1.347727651 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O2 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod2
Violación de segmento ('core' generado)
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O2 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod2
Violación de segmento ('core' generado)
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O2 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod2
Tiempo(seg.):1.332186459 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O0 -g prod1.c -o prod1 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod1
Tiempo(seg.):4.809468198 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O1 -g prod1.c -o prod1 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod1
Tiempo(seg.):1.322256883 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O2 -g prod1.c -o prod1 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod1
Tiempo(seg.):1.329587651 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O3 -g prod1.c -o prod1 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod1
Tiempo(seg.):1.319365308 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O5 -g prod1.c -o prod1 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod1
Tiempo(seg.):3.805547672 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O0 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod2
Tiempo(seg.):4.732511589 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O1 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod2
Tiempo(seg.):1.199007074 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O1 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O2 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O1 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O2 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod2
Tiempo(seg.):1.307102188 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ gcc -O3 -g prod2.c -o prod2 -lrt
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$ ./prod2
Tiempo(seg.):0.577169139 / Tamaño Vectores:1000
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/producto$

```

```
rjraya@rjraya-SATELLITE-C55-A-1EL: ~/Documentos/ac/practicas/BloquePractico4/produ
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O1 -S prod1.c -o prod11.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O1 -S prod0.c -o prod01.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O2 -S prod0.c -o prod02.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O3 -S prod0.c -o prod03.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O1 -S prod0.c -o prod01.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O1 -m32 -S prod0.c -o prod01.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O2 -m32 -S prod0.c -o prod02.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O3 -m32 -S prod0.c -o prod03.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O1 -m32 -S prod1.c -o prod11.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O2 -m32 -S prod1.c -o prod12.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$ gcc -O3 -m32 -S prod1.c -o prod13.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/p
ucto$
```



**B) CÓDIGO FIGURA 1:**

Mostramos primeramente el código original que se propone:

**CÓDIGO FUENTE:** bench1.c

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

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

main()
{
    //Array con los resultados. Orden de preferencia: local, global, puntero.
    int R[40000];
    int ii,i;
    int X1,X2;
    ...

    clock_gettime(CLOCK_REALTIME,&cgt1);
    for (ii=1; ii<=40000;ii++) { //tarda más si quito el =
        X1=0; X2=0;
        for(i=0; i<5000;i+=3){
            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;
        } //fusión de bucles (no hay dependencias)

        if (X1<X2) R[ii]=X1; else R[ii]=X2;
    }
    clock_gettime(CLOCK_REALTIME,&cgt2);
    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double) ((cgt2.tv_nsec-
cgt1.tv_nsec)/(1.e+9));
    printf("Tiempo(seg.):%11.9f \n",ncgt);
}
```

Incluyo como ahora la mejor versión en términos de tiempos de ejecución que se corresponde con el código bench3.c:

**CÓDIGO FUENTE:** bench3.c

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

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

main()
{
```

```

//Array con los resultados. Orden de preferencia: local, global, puntero.
int R[40000];
int ii,i;
int X1,X2;
...

clock_gettime(CLOCK_REALTIME,&cgt1);
for (ii=1; ii<=40000;ii++) { //tarda más si quito el =
    X1=0; X2=0;
    for(i=0; i<5000;i+=3){
        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;
    } //fusión de bucles (no hay dependencias)

    if (X1<X2) R[ii]=X1; else R[ii]=X2;
}
clock_gettime(CLOCK_REALTIME,&cgt2);
ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double) ((cgt2.tv_nsec-
cgt1.tv_nsec)/(1.e+9));
printf("Tiempo(seg.):%11.9f \n",ncgt);
}

```

Finalmente, otra de las versiones probadas:

#### **CÓDIGO FUENTE:** bench5.c

```

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

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

main()
{
    //Array con los resultados. Orden de preferencia: local, global, puntero.
    int R[40000];
    int ii,i;
    int X1,X2;
    ...

    clock_gettime(CLOCK_REALTIME,&cgt1);
    for (ii=1; ii<=40000;ii++) { //tarda más si quito el =
        X1=0; X2=0;
        for(i=0; i<5000;i+=3){
            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;
        } //fusión de bucles (no hay dependencias)

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

```

```

clock_gettime(CLOCK_REALTIME,&cgt2);
ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double) ((cgt2.tv_nsec-
cgt1.tv_nsec)/(1.e+9));
printf("Tiempo(seg.):%11.9f \n",ncgt);
}

```

### MODIFICACIONES REALIZADAS:

Las modificaciones que considero definitivas corresponden a los programas bench3 y bench5.

#### Modificación a) –explicación–:

En la modificación que aparece en el bench3 se ha fusionado el bucle interno y se ha desenrollado en cuatro operaciones por iteración, que por alguna razón era lo óptimo para nuestra máquina.

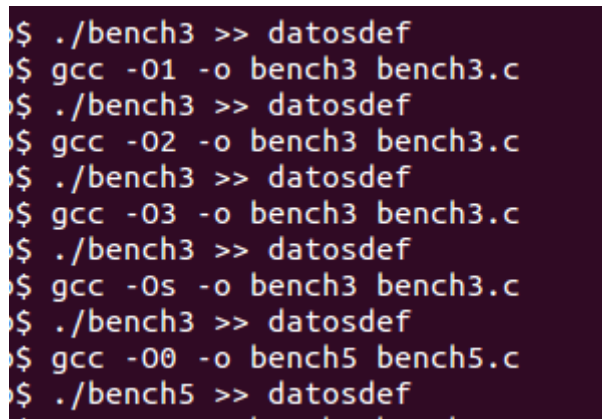
#### Modificación b) –explicación–:

En esta modificación aparte de lo anterior hemos sustituido las operaciones de multiplicación por desplazamientos u operaciones aritméticas.

...

Modificación	-O0	-O1	-O2	-O3	-Os
Sin modificar	1.294866658	0.395386469	0.379327801	0.190251608	0.383353143
Modificación a)	0.675545839	0.094762812	0.076610281	0.076584705	0.081782660
Modificación b)	0.676721683	0.098872534	0.076620821	0.076676458	0.080218209

### CAPTURAS DE PANTALLA:



```

$ ./bench3 >> datosdef
$ gcc -O1 -o bench3 bench3.c
$ ./bench3 >> datosdef
$ gcc -O2 -o bench3 bench3.c
$ ./bench3 >> datosdef
$ gcc -O3 -o bench3 bench3.c
$ ./bench3 >> datosdef
$ gcc -Os -o bench3 bench3.c
$ ./bench3 >> datosdef
$ gcc -O0 -o bench5 bench5.c
$ ./bench5 >> datosdef

```

```
rjraya@rjraya-SATELLITE-C55-A-1EL: ~/Documentos/ac/practicas/BloquePractico4/bench
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O3 -m32 -S bench5.c -o bench53.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O2 -m32 -S bench5.c -o bench52.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O2 -m32 -S bench5.c -o bench52.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O1 -m32 -S bench5.c -o bench51.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O0 -m32 -S bench5.c -o bench50.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -Os -m32 -S bench5.c -o bench5s.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O0 -m32 -S bench3.c -o bench30.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O1 -m32 -S bench3.c -o bench31.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O2 -m32 -S bench3.c -o bench32.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -O3 -m32 -S bench3.c -o bench33.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$ gcc -Os -m32 -S bench3.c -o bench3s.s
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/bench$
```

**COMENTARIOS SOBRE LOS RESULTADOS:**

Hemos sido testigos del poder optimizador de gcc que eliminaba todo el trabajo si no llegamos a añadir dos printf después de la llamada a clock\_gettime. Por otro lado aunque hemos esforzado para que algunas de las multiplicaciones sean sustituidas por productos no hemos conseguido en este sentido buenos resultados. Examinemos los códigos ensamblador para razonar por qué ha sido así:

**Ensamblador versión b) optimización 3**

all	clock_gettime
	movl \$4, %edx
	.p2align 4,,7
	.p2align 3
.L7:	
	xorl %ecx, %ecx
	xorl %esi, %esi
	xorl %eax, %eax
	.p2align 4,,7
	.p2align 3
.L5:	
	movl s(%eax,8), %ebx
	leal (%edx,%ebx,2), %edi
	movl s+4(%eax,8), %ebx
	addl %edi, %esi
	leal (%ebx,%ebx,2), %edi
	movl s+8(%eax,8), %ebx
	subl %edx, %edi
	addl %edi, %ecx
	leal (%edx,%ebx,2), %edi
	movl s+12(%eax,8), %ebx
	addl %esi, %edi
	leal (%ebx,%ebx,2), %esi
	subl %edx, %esi
	addl %ecx, %esi
	movl s+16(%eax,8), %ecx
	leal (%edx,%ecx,2), %ecx
	addl %ecx, %edi
	movl s+20(%eax,8), %ecx
	leal (%ecx,%ecx,2), %ebx
	movl s+24(%eax,8), %ecx
	subl %edx, %ebx
	addl %esi, %ebx
	leal (%edx,%ecx,2), %esi
	addl %edi, %esi
	movl s+28(%eax,8), %edi
	addl \$4, %eax
	leal (%edi,%edi,2), %ecx
	subl %edx, %ecx
	addl %ebx, %ecx
	cmpl \$5000, %eax
	jne .L5
	addl \$4, %edx
	cmpl \$40004, %edx
	jne .L7
	leal 40(%esp), %eax
	movl %eax, 4(%esp)
	movl \$0, (%esp)
	movl %ecx, 24(%esp)
	call clock_gettime

## Ensamblador versión b) optimización 3

```

call    clock_gettime
movl    $4, %edx
.p2align 4,,7
.p2align 3
.L7:
xorl    %ecx, %ecx
xorl    %esi, %esi
xorl    %eax, %eax
.p2align 4,,7
.p2align 3
.L5:
movl    s(,%eax,8), %ebx
leal    (%edx,%ebx,2), %edi
movl    s+4(,%eax,8), %ebx
addl    %edi, %esi
leal    (%ebx,%ebx,2), %edi
movl    s+8(,%eax,8), %ebx
subl    %edx, %edi
addl    %edi, %ecx
leal    (%edx,%ebx,2), %edi
movl    s+12(,%eax,8), %ebx
addl    %esi, %edi
leal    (%ebx,%ebx,2), %esi
subl    %edx, %esi
addl    %ecx, %esi
movl    s+16(,%eax,8), %ecx
leal    (%edx,%ecx,2), %ecx
addl    %ecx, %edi
movl    s+20(,%eax,8), %ecx
leal    (%ecx,%ecx,2), %ebx
movl    s+24(,%eax,8), %ecx
subl    %edx, %ebx
addl    %esi, %ebx
leal    (%edx,%ecx,2), %esi
addl    %edi, %esi
movl    s+28(,%eax,8), %edi
addl    $4, %eax
leal    (%edi,%edi,2), %ecx
subl    %edx, %ecx
addl    %ebx, %ecx
cmpl    $5000, %eax
jne     .L5
addl    $4, %edx
cmpl    $40004, %edx
jne     .L7
leal    40(%esp), %eax
movl    %eax, 4(%esp)
movl    $0, (%esp)
movl    %ecx, 24(%esp)
call    clock_gettime

```

En ambos casos observamos que no hay ni una sola multiplicación, gcc ya se encarga de hacer estas optimizaciones. En concreto las multiplicaciones por 3 se implementan con leal.

Es interesante también discutir las diferencias en ensamblador que se observan entre la primera y la última versión:

#### Ensamblador versión original optimización 2

call	clock_gettime
movl	\$1, %ecx
	.p2align 4,,7
	.p2align 3
.L4:	
	xorl %ebx, %ebx
	xorl %eax, %eax
	.p2align 4,,7
	.p2align 3
.L9:	
	movl s(%eax,8), %edx
	addl \$1, %eax
	leal (%ecx,%edx,2), %edx
	addl %edx, %ebx
	cmpl \$5000, %eax
	jne .L9
	xorl %esi, %esi
	xorw %ax, %ax
	.p2align 4,,7
	.p2align 3
.L7:	
	movl s+4(%eax,8), %edx
	addl \$1, %eax
	leal (%edx,%edx,2), %edx
	subl %ecx, %edx
	addl %edx, %esi
	cmpl \$5000, %eax
	jne .L7
	addl \$1, %ecx
	cmpl \$40001, %ecx
	jne .L4
	leal 40(%esp), %eax
	movl %eax, 4(%esp)
	movl \$0, (%esp)
	call clock_gettime

En este momento, comparando esta versión con la posterior, caemos en la cuenta de por qué un desenrollado puede llegar a ser mucho más eficiente que el bucle usual. Primero porque se ahorran tareas de control, pero también porque se puede aprovechar mejor la localidad de los datos y aunque supongo que el compilador optimiza los saltos del bucle creo que es mejor la nueva versión porque provoca muchos menos saltos.

## Ensamblador versión original optimización 2

```

call    clock_gettime
        movl    $4, %edx
        .p2align 4,,7
        .p2align 3
.L7:
        xorl    %ecx, %ecx
        xorl    %esi, %esi
        xorl    %eax, %eax
        .p2align 4,,7
        .p2align 3
.L5:
        movl    s(,%eax,8), %ebx
        leal    (%edx,%ebx,2), %edi
        movl    s+4(,%eax,8), %ebx
        addl    %edi, %esi
        leal    (%ebx,%ebx,2), %edi
        movl    s+8(,%eax,8), %ebx
        subl    %edx, %edi
        addl    %edi, %ecx
        leal    (%edx,%ebx,2), %edi
        movl    s+12(,%eax,8), %ebx
        addl    %esi, %edi
        leal    (%ebx,%ebx,2), %esi
        subl    %edx, %esi
        addl    %ecx, %esi
        movl    s+16(,%eax,8), %ecx
        leal    (%edx,%ecx,2), %ecx
        addl    %ecx, %edi
        movl    s+20(,%eax,8), %ecx
        leal    (%ecx,%ecx,2), %ebx
        movl    s+24(,%eax,8), %ecx
        subl    %edx, %ebx
        addl    %esi, %ebx
        leal    (%edx,%ecx,2), %esi
        addl    %edi, %esi
        movl    s+28(,%eax,8), %edi
        addl    $4, %eax
        leal    (%edi,%edi,2), %ecx
        subl    %edx, %ecx
        addl    %ebx, %ecx
        cmpl    $5000, %eax
        jne     .L5
        addl    $4, %edx
        cmpl    $40004, %edx
        jne     .L7
        leal    40(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $0, (%esp)
        movl    %ecx, 24(%esp)
        call    clock_gettime

```



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 acarrearán. Incorpore los códigos al cuaderno de prácticas y destaque las diferencias entre ellos.
- (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 para la familia y modelo de procesador que está utilizando) y compárela con el valor obtenido para Rmax. -Consulte la Lección 3 del Tema 1.

### CÓDIGO FUENTE: daxpy.c

```
#include <stdio.h>
#include <math.h>
#include <time.h>
long N = 150000000;
long y[150000000],x[150000000], a,i;

int main()
{
    double ncgt;
    struct timespec cgt1,cgt2;
    clock_gettime(CLOCK_REALTIME,&cgt1);
    //Algoritmo
    for (i=0;i<N;i++){ y[i]=y[i]+a*x[i];}

    clock_gettime(CLOCK_REALTIME,&cgt2);
    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double) ((cgt2.tv_nsec-
cgt1.tv_nsec)/(1.e+9));
    printf("Tiempo(seg.):%11.9f \n",ncgt);
    return 0;
}
```

Tiempos ejec.	-O0	-O1	-O2	-O3	-Os
	0.648178504	0.326869672	0.308875833	0.318300242	0.327565422

**CAPTURAS DE PANTALLA:**

```

rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -O0 -o daxpy0 daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -O1 -o daxpy1 daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -O2 -o daxpy2 daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -O3 -o daxpy3 daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -Os -o daxpys daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpy0
Tiempo(seg.):0.648178504
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpy1
Tiempo(seg.):0.326869672
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpy2
Tiempo(seg.):0.309593152
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpy2
Tiempo(seg.):0.308875833
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpy3
Tiempo(seg.):0.317701432
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpy3
Tiempo(seg.):0.319288218
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpy3
Tiempo(seg.):0.318643826
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpy3
Tiempo(seg.):0.318300242
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpys
Tiempo(seg.):0.323845217
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ./daxpys
Tiempo(seg.):0.327565422

```

```

rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -m32 -O1 -S -o daxpy1.s daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -m32 -O2 -S -o daxpy2.s daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -m32 -O3 -S -o daxpy3.s daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ gcc -m32 -Os -S -o daxpys.s daxpy.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico4/daxpy$ ls
daxpy0.s daxpy1.s daxpy2.s daxpy3.s daxpy.c daxpys.s ejecutables

```

## COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:

### Versión O0

Apenas hay transformaciones frente al código original, tan sólo se ha realizado la traducción.

Hay que decir que en esta versión el número de movs que realiza el bucle resulta muy perjudicial. De hecho hay secuencias totalmente inexplicables por ejemplo:

```
movl  i, %eax      //i
movl  i, %edx
movl  y(%edx,4), %ecx //y[i]
movl  i, %edx
movl  x(%edx,4), %ebx //x[i]
```

donde se llega a captar tres veces el operando i cuando con una habría bastado.

### Versión O1

Se trata de transformaciones que preservan el orden de ejecución.

Aquí aparecen ya algunas mejores por ejemplo en la gestión de saltos, instrucciones de movimiento condicional como `cmovle %eax, %ecx`. Es interesante observar como sin embargo algunos saltos incondicionales se sustituyen por saltos condicionales como en:

```
movl  N, %ecx
testl %ecx, %ecx
jle   .L2
movl  a, %ebx
movl  $0, %eax
```

quizás se intenta hacer algunas de las operaciones de memoria especulativamente ya que en la anterior versión:

```
movl  $0, i
jmp   .L2
```

sería más costoso a nivel hardware porque hay que precaptar instrucciones que se encuentran en la dirección del salto.

Siguiendo observando la situación encontramos la razón por la cual está versión es mucho más eficiente que la anterior. En este caso el cuerpo del bucle ocupa 5 instrucciones mientras que en el anterior eran 15.

### Version O2

Se trata de transformaciones más agresivas que pueden afectar al orden de ejecución generando un código más rápido.

Algunos cambios que llaman la atención serían los siguientes `.p2align 4,,7` `.p2align 3`, estas instrucciones están documentadas aquí:

<https://sourceware.org/binutils/docs/as/P2align.html#P2align>.

Entiendo que se está alineando el propio código de modo que la nueva sección empiece en un múltiplo de cuatro bytes saltándose como máximo 7 bytes (en otro caso no tiene sentido) y si se salta 7 bytes prefiere alinearlos a un múltiplo de 8 bytes. Este alineamiento sirve para mejorar las faltas de caché por el principio de localidad espacial se consigue que las secciones queden en la misma línea de caché.

Otro detalle interesante es el modo de acceso al array, en la versión O1:

```
movl    %ebx, %edx    //a
imull   x(%eax,4), %edx //x[i]
addl    %edx, y(%eax,4) //y[i]
addl    $1, %eax      //i++
```

mientras que en la versión O2:

```
movl    x(%eax), %edx
imull    %ebx, %edx
addl    %edx, y(%eax)
addl    $4, %eax
```

observamos que en el primer caso se trataba como un índice y aquí ha preferido sumar los 4 bytes que determina un entero ahorrándose hacer la cuenta  $4 * \text{eax}$  aunque se supone que esto implementado con desplazamientos, la nueva versión ahorra algunas cuentas. Sin embargo, se ha tenido que hacer :

```
leal    0(%esi,4), %ecx
```

para operar con N en términos de bytes, previamente.

### Versión O3

Se trata de transformaciones muy agresivas que pueden o no proveer de mejor código el orden de ejecución es completamente distorsionado. La situación en este ensamblador es sorprendente, quizá se trate de algún metadato pero es exactamente igual que el ensamblador 2 y tarda unas milésimas más.

### Versión Os

Optimización para tamaño. Esta opción permite transformaciones que reducen el tamaño del código generado. En ocasiones esto ayuda a ejecutar más rápido porque hay menos faltas de página y menos código que ejecutar.

Se observa que se deshacen algunos cambios por ejemplo volviendo al modo de direccionamiento  $x(\text{eax},4)$  y se omiten los `align` antes mencionados. El pequeño decremento de velocidad podría deberse a que en cada iteración del bucle de esta versión se maneja un salto condicional y otro incondicional mientras que en la versión 3 sólo se manejaba el condicional.

## CÓDIGO EN ENSAMBLADOR:

daxpy00.s

	call	clock_gettime
	movl	\$0, i
	jmp	.L2
.L3:	movl	i, %eax
	movl	i, %edx
	movl	y(,%edx,4), %ecx
	movl	i, %edx
	movl	x(,%edx,4), %ebx
	movl	a, %edx
	imull	%ebx, %edx
	addl	%ecx, %edx
	movl	%edx, y(,%eax,4)
	movl	i, %eax
	addl	\$1, %eax
	movl	%eax, i
.L2:	movl	i, %edx
	movl	N, %eax
	cmpl	%eax, %edx
	j1	.L3
	leal	40(%esp), %eax
	movl	%eax, 4(%esp)
	movl	\$0, (%esp)
	call	clock_gettime

daxpy01.s

	call	clock_gettime
	movl	\$0, i
	movl	N, %ecx
	testl	%ecx, %ecx
	jle	.L2
	movl	a, %ebx
	movl	\$0, %eax
.L4:	movl	%ebx, %edx
	imull	x(,%eax,4), %edx
	addl	%edx, y(,%eax,4)
	addl	\$1, %eax
	cmpl	%ecx, %eax
	j1	.L4
	testl	%ecx, %ecx
	movl	\$1, %eax
	cmovle	%eax, %ecx
	movl	%ecx, i
.L2:	leal	40(%esp), %eax
	movl	%eax, 4(%esp)
	movl	\$0, (%esp)
	call	clock_gettime

## daxpy02.s

```

call    clock_gettime
        movl    N, %esi
        xorl    %eax, %eax
        movl    $0, i
        movl    a, %ebx
        testl   %esi, %esi
        leal    0(,%esi,4), %ecx
        jle     .L4
        .p2align 4,,7
        .p2align 3
.L6:
        movl    x(%eax), %edx
        imull   %ebx, %edx
        addl    %edx, y(%eax)
        addl    $4, %eax
        cmpl    %ecx, %eax
        jne     .L6
        movl    %esi, i
.L4:
        leal    40(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $0, (%esp)
        call    clock_gettime

```

## daxpy03.s

```

call    clock_gettime
        movl    N, %esi
        xorl    %eax, %eax
        movl    $0, i
        movl    a, %ebx
        testl   %esi, %esi
        leal    0(,%esi,4), %ecx
        jle     .L4
        .p2align 4,,7
        .p2align 3
.L6:
        movl    x(%eax), %edx
        imull   %ebx, %edx
        addl    %edx, y(%eax)
        addl    $4, %eax
        cmpl    %ecx, %eax
        jne     .L6
        movl    %esi, i
.L4:
        leal    40(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $0, (%esp)
        call    clock_gettime

```

```

daxpy0s.s
call    clock_gettime
        movl    N, %edx
        addl    $16, %esp
        movl    a, %ecx
        xorl    %eax, %eax
.L2:
        cmpl    %edx, %eax
        jge     .L6
        movl    x(,%eax,4), %ebx
        imull   %ecx, %ebx
        addl    %ebx, y(,%eax,4)
        incl    %eax
        jmp     .L2
.L6:
        movl    %eax, i
        pushl   %eax
        pushl   %eax
        leal    -16(%ebp), %eax
        pushl   %eax
        pushl   $0
        call    clock_gettime

```

**Ejercicio extra:**

Los tiempos para datos en coma flotante son:

Tiempos ejec.	-O0	-O1	-O2	-O3	-Os
	0.000227908	0.000148577	0.000151444	0.000121943	0.000151718

La mejor versión se da con optimización O3. Consultando las tablas de intel de la bibliografía vemos que la operación addsd tarda 5 ciclos y mulsd 7 ciclos

Número de elementos	GFLOPS	Tiempo
1000	0.36	0.000005606
2000	0.29	0.000013802
3000	0.26	0.000023249
4000	0.25	0.000032083
5000	0.24	0.000042042
6000	0.21	0.000056736
7000	0.21	0.000066658
8000	0.21	0.000075798
9000	0.22	0.000083172
10000	0.21	0.000094575
11000	0.22	0.000097886
12000	0.22	0.000108433

13000	0.21	0.000123466
14000	0.21	0.000131197
15000	0.24	0.000124291
16000	0.22	0.000146298

Parece que hemos errado en la disposición de la tabla según el comportamiento que aparece en la lección 3. La rehacemos para distintos tamaños:

Número de elementos	GFLOPS	Tiempo (s)
250	0.59	0.000000842
500	0.18	0.000005275
750	0.28	0.000005834
1000	0.33	0.000005987
1250	0.41	0.000006104
1500	0.29	0.000010233
1750	0.24	0.000014383
2000	0.28	0.000014066
2250	0.30	0.000014602
2500	0.26	0.000019110
2750	0.24	0.000022939
3000	0.26	0.000023203

Según estos datos  $R_{max}$  se situaría en 0.40 GFLOPS y se conseguiría para 2500 operaciones en coma flotante.  $N1/2$  se situaría alrededor de las 1000 operaciones en coma flotante.

Por otro lado, con los datos extraídos de las tablas de intel considerando sólo operaciones de suma y producto en coma flotante. Entonces en un ciclo lo mejor posible sería realizar sumas y se podrían realizar hasta 0.2 sumas es decir al quinta parte de una suma. Teniendo en cuenta que la frecuencia del procesador es 2.40 GHz resulta una velocidad pico de 0.48 Gflops. Esto es buena señal de que nuestro experimento ha ido bien ya que no se aleja demasiado de  $R_{max}$  y es superior a él. Todo esto encaja con el comportamiento de la gráfica de la diapositiva 82 del tema 3.