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): Emilio Chica Jiménez
Grupo de prácticas:B2
Fecha de entrega:
Fecha evaluación en clase:

Versión de gcc utilizada: *(respuesta)*

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: `pmm-secuencial-modificado.c`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
//#include <omp.h>
//#define PRINTF_ALL

main(int argc, char **argv)
{
            if(argc < 2) {
                        fprintf(stderr,"Falta fila y columna\n");
                        exit(-1);
            }

            struct timespec cgt1,cgt2; double ncgt; //para tiempo de
```

```
ejecución
              int i,k, n = atoi(argv[1]),h,j,kk,jj;
              int bsize=64;
              double sumalocal=0;
              double **m;
              double **m2;
              double **res;
              m = (double**)malloc(n*sizeof(double*));
              m2 = (double**)malloc(n*sizeof(double*));
              res = (double**)malloc(n*sizeof(double*));
              int en = bsize * (n/bsize); /* Amount that fits evenly into
blocks */


              //Reservo el espacio para las matrices
              for(i=0;i<n;++i)
                            m[i]=(double*)malloc(n*sizeof(double));
              for(i=0;i<n;++i)
                            m2[i]=(double*)malloc(n*sizeof(double));
              for(i=0;i<n;++i)
                            res[i] = (double*)malloc(n*sizeof(double));


              //Inicializo la matriz m y m2
              clock_gettime(CLOCK_REALTIME,&cgt1);
              for (i=0; i<n; i++){

                            for(k=0;k<n;++k){
                                          m[i][k]=2;
                                          m2[i][k]=3;
                                          res[i][k]=0;
                            }

              }
              clock_gettime(CLOCK_REALTIME,&cgt2);
              ncgt+=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double)
((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));


              //Calculo la multiplicacion de la matriz por la matriz y obtengo
el tiempo
              clock_gettime(CLOCK_REALTIME,&cgt1);
              for (kk = 0; kk < en; kk += bsize) {
                            for (jj = 0; jj < en; jj += bsize) {
                                          for (i = 0; i < n; i++) {
                                                        for (j = jj; j < jj +
bsize; j++) {
                                                                      sumalocal =
m[i][j];
                                                                      for (k =
kk; k < kk + bsize; k++) {

              sumalocal += m[i][k]*m2[k][j];
                                                                      }
                                                                      res[i][j] =
sumalocal;
                                                        }
                                          }
                            }
              }
              clock_gettime(CLOCK_REALTIME,&cgt2);
              ncgt+=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double)
```

```
((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
             //Imprimo los resultados
             #ifdef PRINTF_ALL
             printf("Tiempo(seg.):%11.9f\t / Tamaño Matrices:%u\n",ncgt,n);
             for (h=0; h<n; h++){
                          for (i=0; i<n; i++){
                                       for(k=0;k<n;++k){
                                                    printf("/ m[%d]
[%d]*m2[%d][%d] (%8.6f*%8.6f) + \n",i,k,k,h,m[i][k],m2[k][h]);
                                       }
                                       printf("= res[%i][%i] =%8.2f/
\n",h,i,res[h][i]);
                          }


             }
             #else
        printf("Tiempo(seg.):%11.9f\t / Tamaño Matrices:%u\t/ m[0][0]*m2[0]
[0]=res[0][0](%8.6f+%8.6f=%8.6f) // m[%d][%d]*m2[%d][%d]=res[%d][%d](%8.6f+
%8.6f=%8.6f) /\n", ncgt,n,m[0][0],m2[0][0],res[0][0],n-1,n-1,n-1,n-1,n-1,n-
1,m[n-1][n-1],m2[n-1][n-1],res[n-1][n-1]);
             #endif
             for(i=0;i<n;++i){
                          free(m[i]);
             }

             for(i=0;i<n;++i){

                          free(m2[i]);
             }
             for(i=0;i<n;++i)
                          free(res[i]);
        free(m); // libera el espacio reservado para m
             free(m2); // libera el espacio reservado para m2
             free(res); // libera el espacio reservado para res

}
```

MODIFICACIONES REALIZADAS:

Modificación a) –explicación-:

Mi primera modificación se basa en la localidad de las operaciones y los operandos, para ser mas concreto del operando m[i][j] donde se va a almacenar el resultado. Sabiendo que este operando lo podemos llevar a un registro simplemente utilizando una variable local y con ella evitarnos múltiples operaciones de load y store a memoria y simplemente guardarlas en un registro, aumentamos la velocidad del algoritmo.

Mi segunda modificación es:

He cambiado el bucle general que multiplicaba las matrices por un bucle que multiplica las matrices por bloques, es decir, realizo ($2n^3$ * $3n^2$)/tamaño de bloque. Por lo tanto estoy reduciendo los accesos a memoria principal y realizando las operaciones en memoria cache por lo que gano mucho en velocidad al no tener que acceder a memoria principal. Para calcular el tamaño del bloque he seguido la siguiente regla:

Sabiendo que mi cache L1 es de 128KB y que el tamaño de palabra que uso son 8 bytes porque uso doubles. Puedo calcular el tamaño de bloque con la siguiente formula:

$2*(TamañoBloque)^2 * TamañoPalabra = Cache\ L1.$

Por lo que haciendo calculos da 90,50 pero he usado 64 para que sea múltiplo de 2 y encaje mejor los datos de prueba que ha sido 832 multiplo de 64.

Modificación b) –explicación-:

La diferencia entre los codigos es que las operaciones a medida que se realizan más optimizaciones están ya calculadas y también es que los registros utilizados a partir de la -O1 son todos de 64 bits a diferencia de la -O0 que aun se posiciona con registros de 32bits y accede mucho más a memoria que las optimizaciones -O1,O2,-O3,-Os.

En cada código ensamblador se a incluido comentarios sobre lo que se esta haciendo en ese momento.

Modificación	**-O0**	**-O1**	**-O2**	**-O3**	**-Os**
Sin modificar	*6.380810846*	3.773260694	3.751932832	3.794184087	3.808354406
Modificación a)	**4.745907493**	1.005985805	0.954991991	0.954446367	1.530177613

COMENTARIOS SOBRE LOS RESULTADOS:

La optimización en este caso es prácticamente obligatoria ya que reduce el tiempo de ejecución en 4 veces la sin optimizar. Como podemos ver las modificaciones han echo que nuestro código sea mucho más eficiente y el compilador sepa tratarlo mejor y optimizarlo mejor.

CAPTURAS DE PANTALLA:

B) CÓDIGO FIGURA 1:

CÓDIGO FUENTE: `figura1-modificado.c`

(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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

int main()
{
            struct timespec cgt1,cgt2;
            double ncgt;
            int ii,i;
            double X1,X2,X1_0,X2_0,X1_1,X2_1;
            double *R= (double*)malloc(40000*sizeof(double));
            for(i=0; i<5000;i++){
                        s[i].a = rand()%10;
                        s[i].b = rand()%100;
            }
            clock_gettime(CLOCK_REALTIME,&cgt1);
            for (ii=1; ii<=40000;ii++) {
                        X1=0; X2=0; X1_0=0; X1_1=0; X2_0=0; X2_1=0;
                        for(i=0; i<5000;i+=2){
                                    //X1 desenrrollado
                                    X1_0+=2*s[i].a+ii;
```

```
                                        X1_1+=2*s[i+1].a+ii;
                                        //X2 desenrrollado
                                        X2_0+=3*s[i].b-ii;
                                        X2_1+=3*s[i+1].b-ii;

                            }
                                        X1=X1_0+X1_1;
                                        X2=X2_0+X2_1;

                            /* DEBIDO A QUE NO HE CONSEGUIDO QUE FUNCIONE, HE
QUITADO LA OPTIMIZACIÓN
                            asm(
                                        "xor %rbx,%rbx\n" //HAGO 0 rbx
                                        "movl -104(%rbp), %eax \n" //CARGO ii
a EAX
                                        "cltq \n"//PASO A OPERACIONES DE
64BITS
                                        "leaq 0(,%rax,8), %rdx \n" // ii+8 en
RDX
                                        "movq -64(%rbp), %rax \n" //PASO LA
DIRECCION DE R a rax
                                        "addq %rax, %rdx \n" //ACCEDO A LA
POSICION DE R[ii] sumando al puntero R+ii
                                        "movq -48(%rbp),%rcx \n" // PASO X1 a
RCX
                                        "movq -56(%rbp),%r8 \n" //PASO X2 A r8
                                        "cmp %rcx,%r8 \n" //COMPARO X1,X2
                                        "setge %bl \n" //COMPRUEBO SI EL
RESULTADO ES MAYOR O IGUAL Y LO GUARDO EN %BL
                                        "dec %rbx \n"//dec hace rbx 0xFFFFFFFF
                                        "subq %rcx,%r8 \n" //Meto el resultado
de restar X1-X2 en rcx
                                        "and %rbx,%rcx \n" //Hago el and de
rbx and %rcx
                                        "add %rbx,%r8 \n" //Hago rbx + X2
                            );*/
                            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\t //// s[0] = %8.4f / s[39999] =
%8.4f\n",ncgt,R[0],R[39999]);
                return 0;
}
```

MODIFICACIONES REALIZADAS:

Modificación a) –explicación-:

Las optimizaciones realizadas son un desenrrollado del bucle con el que obtengo una ganancia en velocidad considerada y a partir de desenrrollar más de 2 iteraciones no obtengo ganancia alguna y en lugar de usar 2 bucles he usado un sólo bucle para recorrer la estructura por lo que gano en velocidad al tener en el mismo bucle los accesos a memoria de la estructura. Si la estructura hubiese contenido 2 arrays los dos bucles que estaban puestos anteriormente sería correctos. El tiempo de las optimizaciones realizadas por el compilador aumenta con respecto a la figura1 sin modificar, realmente no tiene mucho sentido que sean peores los resultados.

Modificación	**-O0**	**-O1**	**-O2**	**-O3**	**-Os**
Sin modificar	1.668673226	0.586357668	0.576204739	0.571626446	0.551363544
Modificación a)	**1.115632791**	0.687925050	0.682608661	0.682600560	0.690686853

CAPTURAS DE PANTALLA:

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

   a. 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 acarrean. Incorpore los códigos al cuaderno de prácticas y destaque las diferencias entre ellos.

   b. (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`
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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


int main(){
```

```
              struct timespec cgt1,cgt2;
              double ncgt;
              int i,n;
              double a;

              a=100000;n=100000000;
              double * x = (double*)malloc(n*sizeof(double));
              double * y = (double*)malloc(n*sizeof(double));
              for(i=0;i<n;++i){
                            x[i]=100000;
                            y[i]=100000;
              }
              clock_gettime(CLOCK_REALTIME,&cgt1);
              #pragma unroll(1)
              for (i=0; i<n; 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\t //// y[0] = %8.4f / s[%i] =
%8.4f\n",ncgt,y[0],n-1,y[n-1]);
}
```

	-O0	-O1	-O2	-O3	-Os
Tiempos ejec.	*0.671838508*	0.521335569	0.525574099	0.521416585	0.523239443

CAPTURAS DE PANTALLA:

COMENTARIOS SOBRE LAS DIFERENCIAS EN ENSAMBLADOR:

Las diferencias claras entre O0 y el resto es que utilizan sólo registros para acceder a X y a Y sin necesidad de pila con instrucciones especiales como movapd que aprovechan las funciones de optimización del procesador, trayendo de memoria directamente los valores y volcandolos en los registros de double precisión. Por lo tanto menos accesos a memoria y más rápidos con funciones características del procesador en concreto esas son las diferencias.

CÓDIGO EN ENSAMBLADOR: (ADJUNTAR AL .ZIP)
(LIMITAR AQUÍ EL CÓDIGO INCLUÍDO A LA ZONA DEL CÓDIGO ENSAMBLADOR DONDE SE REALIZA LA OPERACIÓN CON VECTORES)

daxpyO0.s

```
call          clock_gettime
              movl          $0, -72(%rbp) //I=0
              jmp           .L4
.L5:
              movl          -72(%rbp), %eax //I A EAX
              cltq
              leaq          0(,%rax,8), %rdx //RDX = I+8
              movq          -48(%rbp), %rax  // Y A RAX
              addq          %rdx, %rax   //RDX = I+8 +Y
              movl          -72(%rbp), %edx //I= I+8 +Y
              movslq        %edx, %rdx   //RDX =EDX
              leaq          0(,%rdx,8), %rcx //RCX = I+8
              movq          -48(%rbp), %rdx  //RDX= Y
              addq          %rcx, %rdx   //RCX =I+8+Y
              movsd         (%rdx), %xmm1//VALOR DE RDX A XMM1
```

```
            movl        -72(%rbp), %edx          //EDX=I
            movslq      %edx, %rdx   //RDX = EDX
            leaq        0(,%rdx,8), %rcx //RCX=I+8
            movq        -56(%rbp), %rdx  //RDX=X
            addq        %rcx, %rdx   //RDX=I+8+X
            movsd       (%rdx), %xmm0//VALOR DE RDX A XMM0
            mulsd       -64(%rbp), %xmm0 //A MULTIPLICO POR XMM0
            addsd       %xmm1, %xmm0 //XMM1 = Y+X*A
            movsd       %xmm0, (%rax)//X A DIRECCION DE RAX
            addl        $1, -72(%rbp)// SUMO 1 A I
.L4:
            movl        -72(%rbp), %eax          //I A EAX
            cmpl        -68(%rbp), %eax          //COMPARO CON N
            jl          .L5                      //FIN DEL BUCLE
            leaq        -16(%rbp), %rax
            movq        %rax, %rsi
            movl        $0, %edi
            call        clock_gettime
```

daxpyO1.s

```
            call        clock_gettime
            movl        $0, %edx          //VARIABLE I
            movsd       .LC0(%rip), %xmm1  //DIRECCION DE X
.L5:
            movapd      %xmm1, %xmm0 //HACE EL LOAD DE LA DIRECCION DE XMM1
A XMM0 POR LO TANTO OBTENGO LA DIRECCION DIRECTAMENTE SIN PASAR POR PILA
            mulsd       0(%rbp,%rdx), %xmm0
            addsd       (%rbx,%rdx), %xmm0
            movsd       %xmm0, (%rbx,%rdx)
            addq        $8, %rdx
            cmpq        $800000000, %rdx //COMPARO CON N
            jne         .L5              ////FIN DEL BUCLE
            leaq        16(%rsp), %rsi
            movl        $0, %edi
            call        clock_gettime
```

daxpyO2.s

```
            call        clock_gettime
            movsd       .LC0(%rip), %xmm1
            xorl        %edx, %edx
            .p2align 4,,10
            .p2align 3
.L5:
            movsd       0(%rbp,%rdx), %xmm0 //NO UTILIZA LA FUNCION MOVAPD
            mulsd       %xmm1, %xmm0
            addsd       (%rbx,%rdx), %xmm0
            movsd       %xmm0, (%rbx,%rdx)
            addq        $8, %rdx
            cmpq        $800000000, %rdx
            jne         .L5               //FIN DEL BUCLE
            leaq        16(%rsp), %rsi
            xorl        %edi, %edi         //OPTIMIZA EL MOVL 0 A EDI DE O1
            call        clock_gettime
```

daxpyO3.s

```
call          clock_gettime
              movq          %rbx, %rdx
              xorl          %r9d, %r9d
              salq          $60, %rdx
              shrq          $63, %rdx
              movapd        (%rsp), %xmm1
              testl         %edx, %edx
              je            .L6
              movsd         .LC0(%rip), %xmm0
              movb          $1, %r9b
              mulsd         0(%rbp), %xmm0
              addsd         (%rbx), %xmm0
              movsd         %xmm0, (%rbx)
.L6:
              movl          $100000000, %eax
              xorl          %ecx, %ecx
              subl          %edx, %eax
              movl          %edx, %edx
              movl          %eax, %r8d
              leaq          0(,%rdx,8), %rsi
              xorl          %edx, %edx
              shrl          %r8d
              leal          (%r8,%r8), %r10d
              leaq          (%rbx,%rsi), %rdi
              addq          %rbp, %rsi
              .p2align 4,,10
              .p2align 3
.L9:
              movsd         (%rsi,%rdx), %xmm0
              addl          $1, %ecx
              movhpd        8(%rsi,%rdx), %xmm0
              mulpd         %xmm1, %xmm0
              addpd         (%rdi,%rdx), %xmm0
              movapd        %xmm0, (%rdi,%rdx)
              addq          $16, %rdx
              cmpl          %r8d, %ecx
              jb            .L9
              cmpl          %r10d, %eax
              leal          (%r9,%r10), %edx
              je            .L8
              movsd         .LC0(%rip), %xmm0
              movslq        %edx, %rdx
              leaq          (%rbx,%rdx,8), %rax
              mulsd         0(%rbp,%rdx,8), %xmm0
              addsd         (%rax), %xmm0
              movsd         %xmm0, (%rax)
.L8:
              leaq          32(%rsp), %rsi
              xorl          %edi, %edi
              call          clock_gettime
```

daxpyOs.s

```
call          clock_gettime
              movsd         .LC0(%rip), %xmm1
              xorl          %edx, %edx
.L5:
              movsd         0(%rbp,%rdx), %xmm0
              mulsd         %xmm1, %xmm0
              addsd         (%rbx,%rdx), %xmm0
              movsd         %xmm0, (%rbx,%rdx)
              addq          $8, %rdx
              cmpq          $800000000, %rdx
```

```
        jne     .L5
        leaq    16(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime
```