Informe de tarea 3

Análisis de rendimiento de código en C mediante técnicas de optimización

Arquitectura de Computadores 2023-2

Profesores Javier Vidal y Juan Felipe González

Integrantes:	**Matrículas:**
Carlos Álvarez Norambuena	2022433621
Nicolás López Cid	2022423927
Sofía López Aguilera	2022451769

Introducción

El presente trabajo busca analizar el impacto de diferentes métodos de optimización vistos en clases en las diferentes funciones propuestas en el enunciado del trabajo. Buscamos comparar en ciclos de reloj el impacto de las diferentes optimizaciones. En cuanto a resultados vamos a ver que para números pequeños en las diferentes funciones no hay un impacto significativo pero para inputs grandes se puede notar una mejora en la cantidad de ciclos de reloj.

Desarrollo

1. Resultados

En esta sección se analizan en detalle las funciones implementadas y los resultados obtenidos para cada caso solicitado por el enunciado.

1.1 Función combine

Esta sección contiene los resultados y análisis de resultados de las preguntas 1-3 del enunciado, abarcando las funciones combine1, combine5 y combine6 contenidas en el archivo combineClock.c. En primer lugar, la función a optimizar era combine1, la cual calculaba la sumatoria o productorio de los números alojados en un arreglo del tamaño que se desee. En nuestro código, al arreglo se le dará solo el input de cuál es el tamaño que se desea. Los números

```
63  // Combine1 no optimizado
64  void combine1(vec_ptr v, data_t *dest) {
65      int i;
66      *dest = IDENT;
67      for (i = 0; i < vec_length(v); i++) {
68          data_t val;
69          get_vec_element(v, i, &val);
70          *dest = *dest OP val;
71      }
72  }
73
```

dentro del arreglo son elegidos al azar, estando en el rango de 1 a 11. El código de la función a optimizar es el siguiente:
Se pueden ver múltiples acciones a tomar para lograr la optimización de este código. Estas se ven reflejadas en las soluciones creadas por nosotros, combine5 y combine6, las cuales implementan "loop unrolling" de 4 vías y además, combine6 resalta por sobre las otras usando paralelismo además de la descomposición iterativa mencionada anteriormente. A continuación, se puede ver el código del método combine5:

```
74  // Combine5 usando loop unrolling con 4 vias
75  void combine5(vec_ptr v, data_t *dest) {
76      long long int i;
77      long long int length = vec_length(v);
78      long long int limit = length - 3;
79      data_t *data = get_vec_start(v);
80      data_t resultado = IDENT;
81
82      for (i = 0; i < limit; i += 4) {
83          resultado = (((resultado OP data[i]) OP data[i + 1]) OP data[i + 2]) OP data[i + 3];
84      }
85
86      for (; i < length; i++) {
87          resultado = resultado OP data[i];
88      }
89
90      *dest = resultado;
91  }
92
```

Notemos las diferencias entre la función combine1 y combine5. Se agregó que las variables se declaran antes de entrar a cualquier bucle, en particular, la llamada a vec_length se guarda en una variable para no hacer la llamada dentro de la estructura *for*. Luego, se puede apreciar que dada cuatro iteraciones del bucle, se aprovecha la misma usando los tres punteros que le siguen a la posición actual, esto es con el fin de aprovechar la misma llamada a la función para ejecutar más de una instrucción a la vez y no tener que acudir a la memoria a partir de distintas direcciones, sino sólo de una (usando aritmética de punteros). Este proceso lo conocemos por "loop unrolling" y cumple con cuatro vías ya que accede a cuatro posiciones de memoria contiguas en el arreglo al mismo tiempo. De esta forma se optimiza bastante el programa.

Aún así, hay una forma de elevar un poco más el rendimiento mediante paralelismo, el cual se ve aplicado en combine6:

```
93  // Combine6 usando loop unrolling con paralelismo y 4-vias
94  void combine6(vec_ptr v, data_t *dest) {
95      long long int i;
96      long long int length = vec_length(v);
97      long long int limit = length - 3;
98      data_t *data = get_vec_start(v);
99      data_t resultado1 = IDENT;
100     data_t resultado2 = IDENT;
101     data_t resultado3 = IDENT;
102     data_t resultado4 = IDENT;
103
104     for (i = 0; i < limit; i += 4) {
105         resultado1= resultado1 OP data[i];
106         resultado2= resultado2 OP data[i + 1];
107         resultado3= resultado3 OP data[i + 2];
108         resultado4= resultado4 OP data[i + 3];
109     }
110
111     for (; i < length; i++) {
112         resultado1 = resultado1 OP data[i];
113     }
114
115     *dest = resultado1 OP resultado2 OP resultado3 OP resultado4;
116 }
```

Esta aplicación difiere en lo visto anteriormente, ya que ahora se definen distintas variables de resultado para eliminar las dependencias que tiene un resultado del otro. De esta forma, usamos pipelining para explotar la capacidad del compilador de hacer varias cosas a la vez sin alterar los datos obtenidos. Finalmente, los resultados se juntan en uno solo operando con la suma o la resta, según se defina OP.

Por otra parte, para el análisis de los datos hemos separado en diversos casos ya que notamos algunos cambios según el tipo de dato en las funciones (entero y flotante) y también según la operación que se ejecutaba (suma o resta). Las tablas a continuación representan las pruebas ejecutadas con casos particulares para cada función:

Cantidad de números aleatorios de 1 a 10 en el arreglo	Promedio de ciclos transcurridos por las funciones combine (suma entera) con 100 test realizados		
	combine1	combine5	combine6
100 números	0.000000	0.000000	0.000000
10000 números	0.100000	0.000000	0.020000
1000000 números	9.680000	1.730000	1.950000
100000000 números	473.170013	92.220001	98.730003

Cantidad de números aleatorios de 1 a 10 en el arreglo	Promedio (en ciclos) transcurridos por las funciones combine (producto entero) con 100 test realizados		
	combine1	combine5	combine6
100 números	0.000000	0.000000	0.000000
10000 números	0.130000	0.010000	0.020000
1000000 números	7.920000	1.690000	1.640000
100000000 números	486.489990	109.320000	108.839996

Cantidad de números aleatorios de 1 a 10 en el arreglo	Promedio de ciclos transcurridos por las funciones combine (suma flotante) con 100 test realizados		
	combine1	combine5	combine6
100 números	0.000000	0.000000	0.000000
10000 números	0.000000	0.010000	0.060000
1000000 números	1.160000	1.760000	5.080000
100000000 números	521.390015	180.860001	120.910004

Cantidad de números aleatorios de 1 a 10 en el arreglo	Promedio de ciclos transcurridos por las funciones combine (producto flotante) con 100 test realizados		
	combine1	combine5	combine6
100 números	0.010000	0.000000	0.000000
10000 números	0.150000	0.050000	0.040000
1000000 números	8.400000	2.550000	1.720000
100000000 números	501.500000	157.440002	110.370003

En primer lugar, se puede notar que el cambio no es muy evidente con arreglos pequeños. Estas optimizaciones se dan de manera más efectiva en arreglos de grandes cantidades de números. Luego, los resultados para la suma entera y producto entero no son lo esperado del todo. La diferencia entre combine1 y las otras dos funciones se nota con evidencia, pero la comparativa entre combine5 y combine6 resulta algo azarosa. Se han hecho numerosas pruebas al respecto y los resultados son los mismos. Ambas versiones son más óptimas de lo esperado, pero a veces una es mejor que otra. En cambio, para cualquier operación hecha con valores flotantes, se nota la diferencia, nuevamente no muy significativa pero presente.

1.2 Función Fibonacci:

Para el siguiente ítem, los códigos descritos se encuentran en el archivo fibonacci.c.

1. Los datos obtenidos de la función de Fibonacci presentada en el enunciado en ciclos de reloj son los siguientes:

Número n que se está calculando de la sucesión de Fibonacci.	Promedio de ciclos transcurridos por las funciones de Fibonacci con 100 test realizados.
	fibonacci
n = 5	0.6 ciclos
n = 10	1.19 ciclos
n = 20	75.580002 ciclos
n = 30	5324.890137 ciclos
n = 40	615397.500000 ciclos

Estos resultados no son de sorprender debido a que el algoritmo recursivo de Fibonacci tiene una complejidad temporal de O(2^n).

2. La optimización que realizamos fue:

```c
long long int fibonacci_opt(long long int *n){
    if (*n == 1) return 0;
    if (*n == 2) return 1;
    if (*n == 3) return 1;

    long long int a = 0, b = 1, c = 1, d = 2;
    long long int i;
    long long int limit = *n - 3;
    for (i = 4; i < limit; i += 4){
        a = c + d;
        b = d + a;
        c = a + b;
        d = b + c;
    }
    for(; i < *n; i++){
        limit = d;
        d = d + c;
        c = limit;
    }

    return d;
}
```

Número n que se está calculando de la sucesión de Fibonacci.	Promedio de ciclos transcurridos por las funciones de Fibonacci con 100 test realizados.	
	fibonacci	fibonacci_opt
n = 5	0.6 ciclos	0.55 ciclos
n = 10	1.19 ciclos	0.59 ciclos
n = 20	75.580002 ciclos	0.6 ciclos
n = 30	5324.890137 ciclos	0.42 ciclos
n = 40	615397.500000 ciclos	0.75 ciclos.

El primer cambio, uno de los más importantes, consiste en implementar Fibonacci de manera iterativa, eliminando así las llamadas recursivas. Esta modificación implica que el programa no

requerirá realizar tantos accesos de memoria a la pila. Además, en esta versión iterativa, hemos aplicado 4 veces la técnica de 'loop unrolling'. Es decir, en cada iteración del bucle for, se calculan 4 números de la sucesión de Fibonacci. Por último, hemos añadido 4 variables (a, b, c, d) al algoritmo en cuestión, donde cada una representa un número de la sucesión. Esta adición se realiza para evitar la repetición constante de las mismas 3 variables, logrando algo similar a esto:

```
for (i = 1; i < limit; i += 4){
    aux = a;
    a = a + b;
    b = aux;
```

De esta manera tratamos de reducir las dependencias entre las variables y que el procesador pueda aprovechar el proceso de *pipelining* mejorando el rendimiento del programa.

Adicionalmente a esto hicimos un experimento complementario para verificar que estas últimas optimizaciones (además del cambiar Fibonacci de recursivo a iterativo), fueran efectivas o no. Escribimos el siguiente código:

```
long long int fibonacci_it(long long int *n){
    long long int a = 1;
    long long int b = 0;
    long long int aux;

    int i;

    for (i = 1; i < *n; i++){
        aux = a;
        a = a + b;
        b = aux;
    }


    return b;
}
```

Realizamos una comparación entre fibonacci_it y fibonacci_opt:

Número n que se está calculando de la sucesión de fibonacci.	Promedio de ciclos transcurridos por las funciones de fibonacci con 100 test realizados.	
	fibonacci_it	fibonacci_opt
n = 20	0.51600 ciclos	0.49100 ciclos
n = 5000	25.04 ciclos	19.02 ciclos
n = 500000	1280 ciclos	970 ciclos.

Después de realizar esta comparación notamos que hay una diferencia entre la versión optimizada y una que solo realiza Fibonacci iterativo por lo que la solución propuesta es efectiva, aunque esta diferencia se hace presente para valores de n grandes. No realizamos las pruebas para la versión recursiva debido a que como esta última es $(O2^n)$tomaría una cantidad enorme de tiempo realizar las pruebas para valores grandes. También vale decir que calcular los valores de Fibonacci para valores tan grandes pierde su sentido, ya que se van a desbordar las variables por tratar de almacenar valores tan grandes, por lo que fue realizado única y exclusivamente para verificar una optimización. Por último, el valor para n = 20 fue calculado de nuevo, por eso difiere del valor en la tabla anterior.

1.3 Función invertir string

Para el siguiente ítem se utilizan los códigos que están en el archivo Ejercicio_6_7.c.

La función creada invertir(), la cual no está optimizada y ocupa el mayor número de abstracciones posible es de la forma:

```
void invertir(char *s) {
    int i;

    for (i = 0; i < strlen(s) - 1; i++) {
        char temp1 = s[i];
        s[i] = s[strlen(s) - 1 - i];
        s[strlen(s) - 1 - i] = temp1;

    }
}
```

La versión optimizada 1, donde ocupamos “loop unrolling” de dos vías para poder ejecutar 2 iteraciones por ciclo, se diferencia en que creamos una variable length, la cual almacena el valor de la longitud de s y además las variables temporales se crean fuera del bucle de la forma:

```
void invertir_optimizado_1(char *s) {
    int length = strlen(s);
    int i;
    char temp1;
    char temp2;

    for (i = 0; i < length - 2; i+=2) {
        temp1 = s[i];
        temp2 = s[i+1];

        s[i] = s[length - 1 - i];
        s[i+1] = s[length - 1 - i];

        s[length - 1 - i] = temp1;
        s[length - 2 - i] = temp2;

    }
}
```

La siguiente es la versión optimizada 2, donde añadimos una iteración más por ciclo, quedando de la siguiente forma:

```c
void invertir_optimizado_2(char *s) {
    int length = strlen(s);
    int i;
    char temp1;
    char temp2;
    char temp3;

    for (i = 0; i < length - 3; i+=3) {
        temp1 = s[i];
        temp2 = s[i+1];
        temp3 = s[i+2];

        s[i] = s[length - 1 - i];
        s[i+1] = s[length - 2 - i];
        s[i+2] = s[length - 3 - i];

        s[length - 1 - i] = temp1;
        s[length - 2 - i] = temp2;
        s[length - 3 - i] = temp3;

    }
}
```

Y finalmente, siguiendo la metodología de las optimizaciones anteriores, la versión optimizada 3 añade una tercera iteración por ciclo, quedando de la siguiente forma:

```
void invertir_optimizado_3(char *s) {
    int length = strlen(s);
    int i;
    char temp1;
    char temp2;
    char temp3;
    char temp4;

    for (i = 0; i < length - 4; i+=4) {
        temp1 = s[i];
        temp2 = s[i+1];
        temp3 = s[i+2];
        temp4 = s[i+3];

        s[i] = s[length - 1 - i];
        s[i+1] = s[length - 2 - i];
        s[i+2] = s[length - 3 - i];
        s[i+3] = s[length - 4 - i];

        s[length - 1 - i] = temp1;
        s[length - 2 - i] = temp2;
        s[length - 3 - i] = temp3;
        s[length - 4 - i] = temp4;

    }
}
```

Para que la cantidad de ciclos fuese significativa, dentro del main se llamó a cada función (por separado) con un bucle for que ejecutaba en total 1.000.001 veces la llamada a cada función, quedando los siguientes resultados:

Cantidad de caracteres de entrada	Versión de la función Invertir String			
	No Optimizada	Optimizada_1	Optimizada_2	Optimizada_3
5 caracteres	87 ciclos	21 ciclos	11 ciclos	11 ciclos
20 caracteres	353 ciclos	77 ciclos	67 ciclos	50 ciclos
100 caracteres	2635 ciclos	315 ciclos	283 ciclos	309 ciclos
500 caracteres	19534 ciclos	735 ciclos	710 ciclos	743 ciclos

En la tabla anterior podemos apreciar que la función invertir string se demora mucho en procesar inputs grandes, mientras que todas las versiones optimizadas ocupan en promedio un 3,5% de los ciclos que ocupa invertir string, lo cual significa que es 26 veces más rápido.

También podemos observar que la mejor versión es Optimizada_2, el segundo lugar está entre las demás versiones optimizadas, ya que Optimizada_3 es mejor cuando se prueba con 5, 20 y 100 caracteres, llegando a ser la más rápida de todas, pero cuando se prueba con 500 caracteres o más la versión Optimizada_1 es superior, por lo que podemos deducir que lo más óptimo es hacer loop unrolling con 2 vías.

Conclusiones

Las optimizaciones implementadas, como el loop unrolling, paralelismo de vías y eliminación de referencias innecesarias a memoria, tuvieron un impacto significativo para cifras grandes en el rendimiento de los programas. Se observó una mejora sustancial en los tiempos de ejecución, validando la eficacia de estas técnicas para optimizar programas.

Por ejemplo, la versión mejor optimizada del procedimiento “Inversa” obtuvo una mejora del 20000% en el rendimiento en comparación con la versión no optimizada. Esta mejora se debe principalmente a la eliminación de redundancias en las referencias a memoria y al uso de loop unrolling con paralelismo de vías.

La comparación entre las versiones original y optimizada del cálculo del n-ésimo término en las funciones también destacó la eficiencia de las estrategias de optimización aplicadas. La eliminación de redundancias en las referencias a memoria y el uso de loop unrolling con paralelismo de vías contribuyeron significativamente a la mejora del rendimiento, subrayando la importancia de considerar estas técnicas en futuros desarrollos.

En general, estos resultados sugieren que las técnicas de optimización utilizadas pueden ser útiles para mejorar el rendimiento de otros programas. Sin embargo, es importante realizar más experimentos para evaluar el rendimiento de estas técnicas en una variedad de programas y conjuntos de datos.