MEMORIA PROYECTO HARDWARE

PRÁCTICA 1

Pablo Moreno Muñoz 841972

Andrés Yubero Segura 842236

ÍNDICE

RESUMEN

En esta práctica se ha trabajado con el juego conectaK, el cual es una extensión del clásico *conecta 4* pero en un tablero de juego sin gravedad y extendido a tener que formar una línea de K fichas en vez de 4. El código ya venía desarrollado en C, a excepción de dos funciones que se han tenido que desarrollar.

Además se ha tenido que ver cómo es el almacenamiento de la estructura interna del juego en memoria y qué uso se hace de la pila desde C, para poder implementar funciones análogas a *conecta_K_hay_linea_c_c* y a *conecta_K_buscar_alineamiento_c* en ensamblador, intentando hacerlas lo más eficiente posible, llegando incluso a eliminar la recursividad de la segunda.

Finalmente, se han creado bancos de pruebas para automatizar los tests con casos críticos, se han tomado medidas de rendimiento para ver cuáles eran las funciones más costosas y se ha analizado el impacto del uso de flags de optimización en el compilador (-O0, -O1, -O2, -O3 y -O3 -Otime).

INTRODUCCIÓN

En esta práctica se ha partido de una base de código en C que ya venía hecha y que permitía ejecutar sobre el microcontrolador LPC2105 el juego conocido como conecta K, lo que ha implicado crear dos funciones en C para depurar con una partida ya inicializada y visualizarla en memoria.

Después, se han creado funciones equivalentes a las que hay en C para buscar alineamiento y comprobar si hay línea en ARM, de manera que se han obtenido distintas implementaciones (funciones de C que llaman a otras de C, de C a ARM, de ARM a C, y de ARM a ARM). Para comprobar que todas estas implementaciones daban el mismo resultado, se ha tenido que crear una función para verificar esta condición.

Posteriormente, se ha hecho una optimización eliminando la recursividad de la función buscar alineamiento en ARM. Finalmente, se han tomado medidas de rendimiento (tamaño del código y tiempos de ejecución) de las diferentes funciones y con distintos niveles de optimización (-O1, -O2, -O3, -O3 -Otime) y comparando estas medidas con las del nivel de optimización -O0.

El funcionamiento del juego se ha explicado en el apartado de resumen. Pero en cuánto a detalles técnicos, cabe mencionar que el tablero se compone de una lista de columnas que almacena los índices de las columnas que para la fila iésima tienen valores no-cero y una lista de no_ceros que almacena los valores de las celdas no vacías.

OBJETIVOS

Los objetivos de este trabajo son los siguientes:

- Interactuar con un microcontrolador y ser capaces de ejecutar y depurar.
- Profundizar en la interacción C / Ensamblador.
- Ser capaces de depurar el código ensamblador que genera un compilador a partir de un lenguaje en alto nivel.
- Saber depurar código siguiendo el estado arquitectónico de la máquina: contenido de los registros y de la memoria.
- Conocer la estructura segmentada en tres etapas del procesador ARM 7, uno de los más utilizados actualmente en sistemas empotrados.
- Familiarizarse con el entorno Keil µVision sobre Windows, con la generación cruzada de código para ARM y con su depuración.
- Aprender a analizar el rendimiento y la estructura de un programa.
- Desarrollar código en ensamblador ARM: adecuado para optimizar el rendimiento.
- Optimizar código: tanto en ensamblador ARM, como utilizando las opciones de optimización del compilador.
- Entender la finalidad y el funcionamiento de las Aplication Binary Interface, ABI, en este caso el estándar ATPCS (ARM-Thumb Application Procedure Call Standard), y combinar de manera eficiente código en ensamblador con código en C.
- Comprobar automáticamente que varias implementaciones de una función mantienen la misma funcionalidad.

METODOLOGÍA

CÓDIGO FUENTE

El código fuente se puede encontrar en el Anexo al final del documento.

Vamos a comentar las cabeceras de las funciones más importantes del código fuente,se explicará cómo funcionan, los parámetros que reciben y dónde los recibe y para qué usa cada registro

conecta_K_buscar_alineamiento_arm

En esta función se cuenta el número de fichas de un mismo color que hay en una dirección y gracias a los deltas comprueba todos los sentidos y direcciones para ver si hay K fichas del mismo color juntas, lo que significa victoria para el jugador.

Parámetros que recibe:

- Dirección inicial del tablero (cuadricula) → r0
- Fila donde se ha colocado la ficha → r1
- Columna donde se ha colocado la ficha → r2
- Color de la ficha introducida → r3
- deltas_fila → se pasa por pila
- deltas_columna → se pasa por pila

Devuelve:

- Número de fichas alineadas

conecta_K_hay_linea_arm

La función comprueba si al introducir una ficha válida en el tablero, el jugador que lo ha introducido ha ganado la partida formando una línea de K fichas

Parámetros que recibe:

- Dirección inicial del tablero (cuadricula) → r0
- Fila → r1
- Columna → r2
- Color de la ficha → r3

Devuelve:

- Un int (0,1) que se usa como booleano que indica si hay línea

MAPA DE MEMORIA

En la imagen, se ve el mapa de memoria generado por el código en C. Para producir este mapa, se empleó el archivo "conecta_K_2023.map," que se encuentra en el directorio de "Listings" después de la compilación del código y contiene información esencial acerca de la memoria.

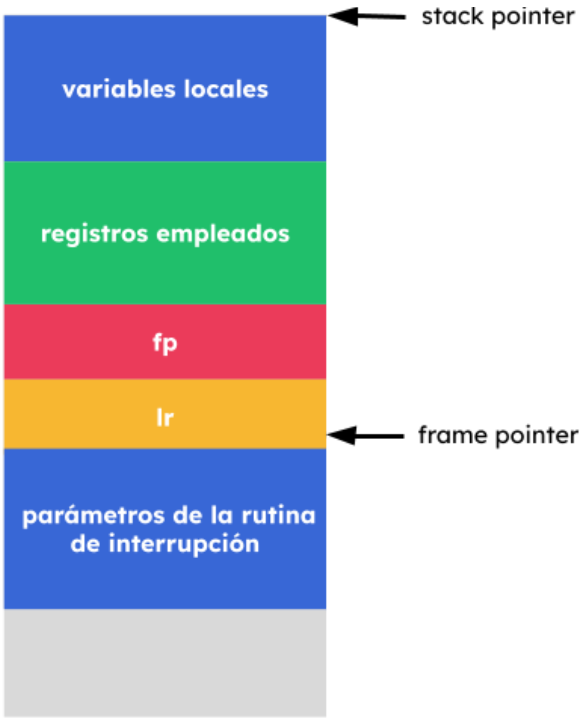
La mayor parte del espacio empleado corresponde al área de la pila (stack), la cual tiene asignados 1160 bytes, abarcando desde la dirección 0x40000578 hasta 0x400000f0. El heap se ubica en la misma dirección, aunque no se le asigna un espacio reservado.

Desde la dirección 0x400000f0 hasta la 0x4000004c, se localiza el espacio destinado a la salida. A partir de esta última dirección hasta la 0x4000003c se encuentra una área dedicada a datos, continuada a esta se encuentra hasta la dirección 0x40000032 la entrada.

Luego llegando hasta la 0x40000001, se encuentra el espacio asignado a la matriz "cuadrícula" ocupa el rango entre las direcciones 0x40000032 y 0x40000000.

Finalmente hay un espacio no utilizado hasta alcanzar la posición de memoria 0x00000da8 En esta ubicación, se concluye la última área relevante de la memoria, que se extiende hasta la dirección 0x0000000, y alberga las instrucciones del programa, es decir, el código del programa en sí.

MARCO ACTIVACIÓN DE LA PILA



En la figura anterior se puede observar el marco de activación de la pila en las funciones desarrolladas en ensamblador ARM.

Más específicamente en las llamadas a funciones como buscar alineamiento tiene 6 parámetros y se almacenan los 4 primeros registros de r0-r3 según el estándar ATPCS. Los deltas filas y columnas se añaden en la pila antes de saltar a la subrutina

También hay que comentar que en ambas funciones se guardan los contenidos de los registros que se utilizarán en la función. (r4-r10, según las especificaciones del ATPCS).

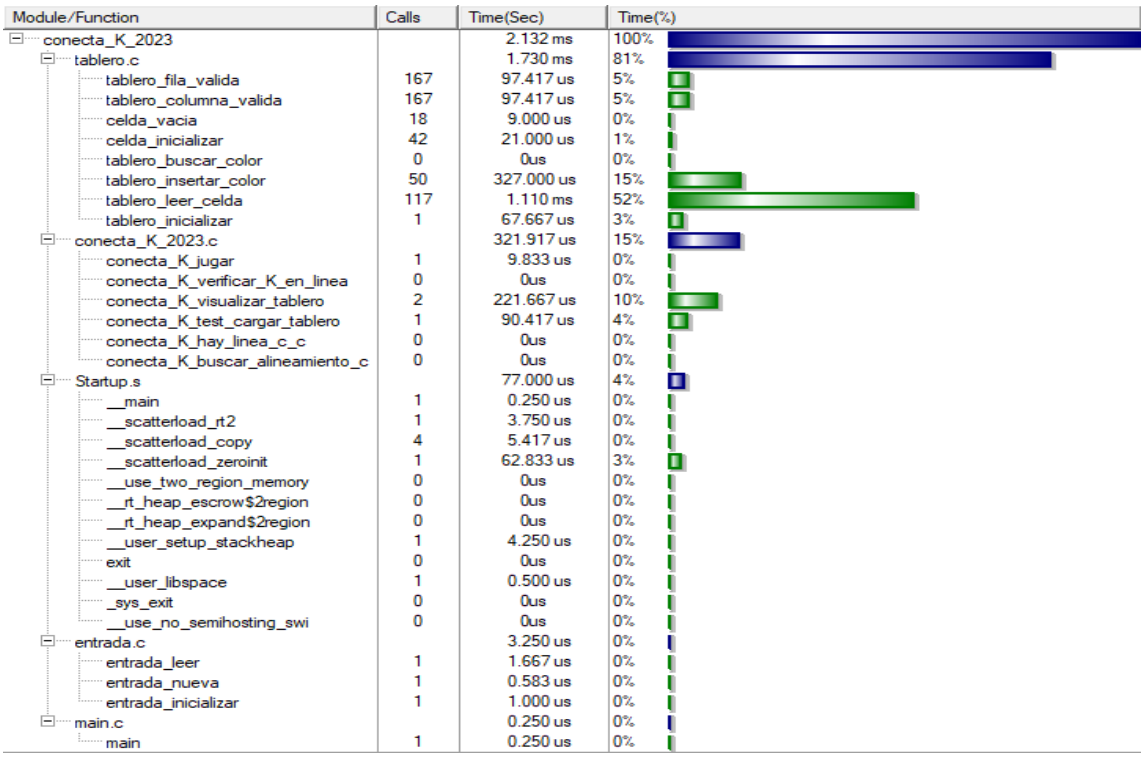
Cabe destacar los registros reservados r14->lr y r11->fp los cuales también se apilan en cada llamada.

El "frame pointer" se sitúa en la dirección anterior a la de los parámetros guardados en la pila, que corresponden a los "deltas." Esto permite acceder a sus valores utilizando las direcciones fp+4 y fp+8.

RESULTADOS

FUNCIONES DE MAYOR PESO

Para analizar el rendimiento, se ha empleado la opción *Performance Analyzer* de Keil, la cual nos ha permitido obtener la siguiente captura con datos sobre las llamadas y tiempo de cada función:



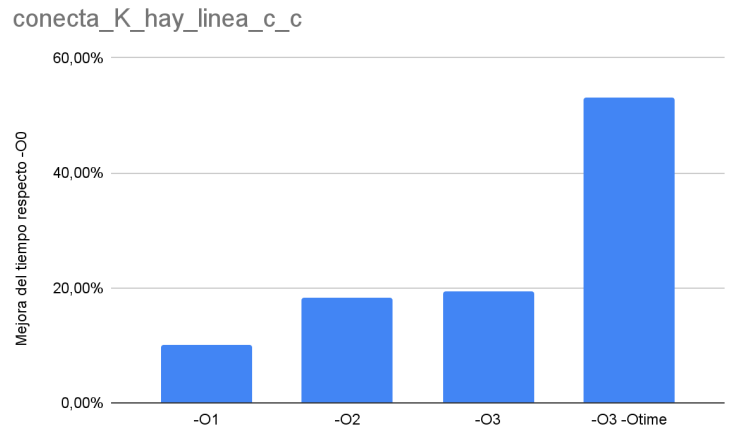
Module/Function	Calls	Time(Sec)	Time(%)
conecta_K_2023		2.132 ms	100%
tablero.c		1.730 ms	81%
tablero_fila_valida	167	97.417 us	5%
tablero_columna_valida	167	97.417 us	5%
celda_vacia	18	9.000 us	0%
celda_inicializar	42	21.000 us	1%
tablero_buscar_color	0	0us	0%
tablero_insertar_color	50	327.000 us	15%
tablero_leer_celda	117	1.110 ms	52%
tablero_inicializar	1	67.667 us	3%
conecta_K_2023.c		321.917 us	15%
conecta_K_jugar	1	9.833 us	0%
conecta_K_verificar_K_en_linea	0	0us	0%
conecta_K_visualizar_tablero	2	221.667 us	10%
conecta_K_test_cargar_tablero	1	90.417 us	4%
conecta_K_hay_linea_c_c	0	0us	0%
conecta_K_buscar_alineamiento_c	0	0us	0%
Startup.s		77.000 us	4%
__main	1	0.250 us	0%
__scatterload_rt2	1	3.750 us	0%
__scatterload_copy	4	5.417 us	0%
__scatterload_zeroinit	1	62.833 us	3%
__use_two_region_memory	0	0us	0%
__rt_heap_escrow$2region	0	0us	0%
__rt_heap_expand$2region	0	0us	0%
__user_setup_stackheap	1	4.250 us	0%
exit	0	0us	0%
__user_libspace	1	0.500 us	0%
_sys_exit	0	0us	0%
__use_no_semihosting_swi	0	0us	0%
entrada.c		3.250 us	0%
entrada_leer	1	1.667 us	0%
entrada_nueva	1	0.583 us	0%
entrada_inicializar	1	1.000 us	0%
main.c		0.250 us	0%
main	1	0.250 us	0%

Como se puede apreciar, la función de mayor coste temporal es *tablero_leer_celda* con un consumo del 52% del cómputo total. Por otro lado en cuanto al número de llamadas *tablero_fila_valida* y *tablero_columna_valida* son las más cargadas con 167 llamadas cada una.

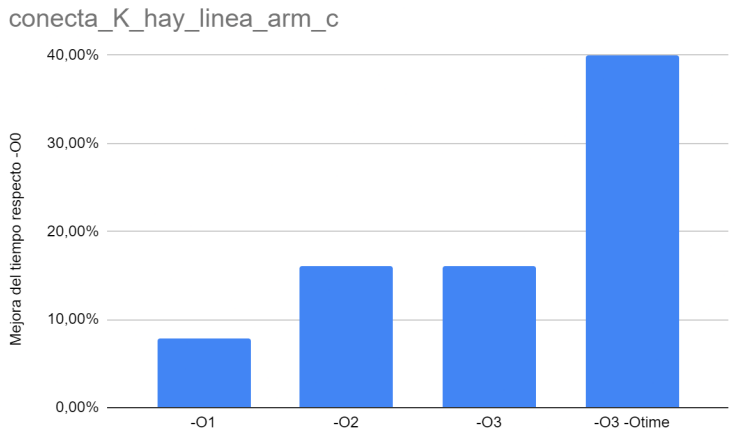
MEDIDAS DE RENDIMIENTO

Para poder obtener las mediciones de rendimiento del compilador (dadas en tiempo de ejecución y tamaño del código), se ha tenido que modificar la función de verificar K en línea para probar cada función de hay línea de manera independiente sin que la medición de alguna afecte al resto. Cabe destacar que todos los porcentajes mostrados en las gráficas están calculados respecto al nivel de optimización -O0.

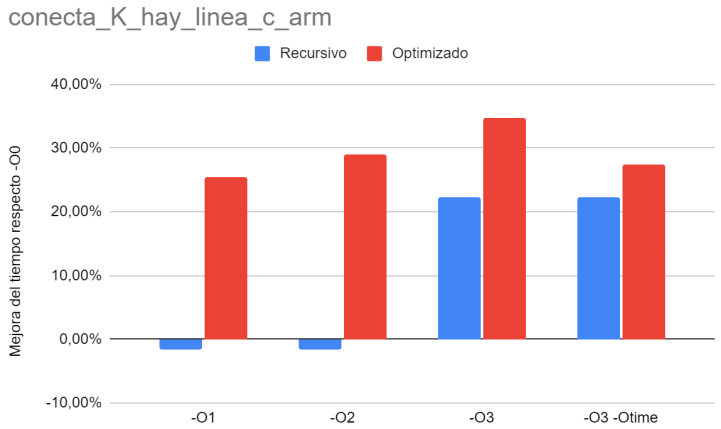
Medidas de tiempo



Como se puede apreciar en el gráfico el nivel de optimización -O3 -Otime es el que bate al resto de niveles de optimización con un 53,08% de mejora respecto al algoritmo -O0.



Al igual que en la gráfica anterior, *en hay_linea_arm_c* -O3 -Otime consigue la mayor mejora en tiempo.

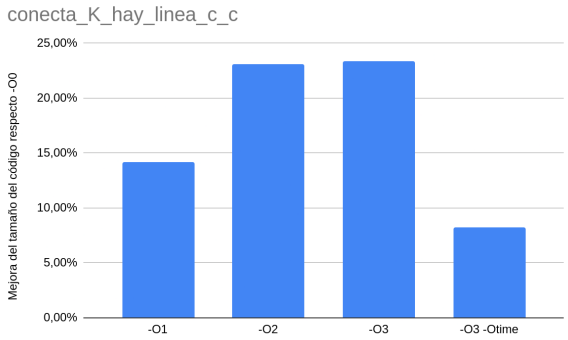


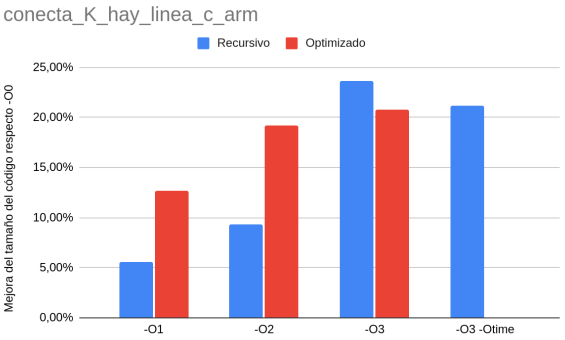
Esta vez la versión no recursiva y con nivel de optimización -O3 consigue ser la más rápida.

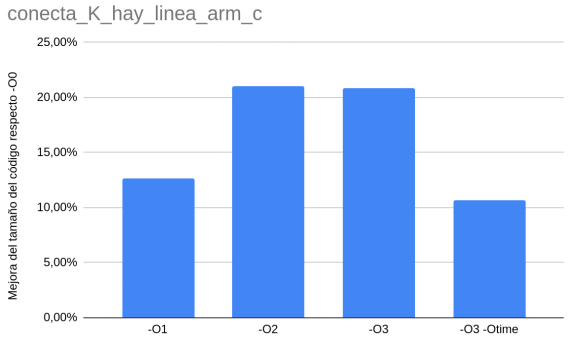
Finalmente, quedarían ver los tiempos de conecta_K_hay_linea_arm_arm, pero como en este caso las funciones que medimos están hechas en ARM, no hay ningún tipo de mejora ya que el compilador de C no es capaz de optimizar ensamblador.

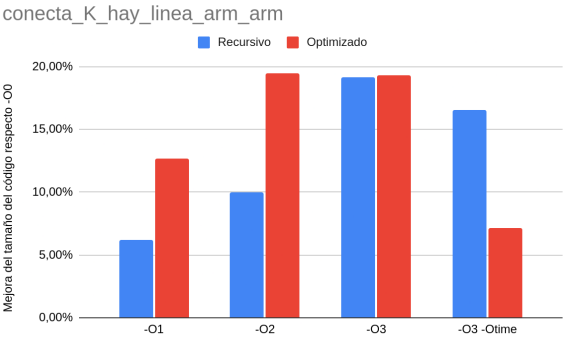
De las tablas con todas las medidas de tiempo que se encuentran en el anexo, podemos sacar como conclusión que, pese a que nuestras dos funciones en arm consiguen tardar apenas 121.833us mientras que el código en C con nivel -O0 tarda 204.583us, finalmente el código en C con nivel -O3 -Otime consigue un mejor rendimiento tardando 95.999us.

Medidas de tamaño del código









Como se puede observar, el código más compacto se obtiene con los niveles de optimización -O2 y -O3. Con -O3 -Otime se prioriza hacer un código que ejecute más rápido, por lo que el código en esos casos queda más grande.

OPTIMIZACIONES REALIZADAS AL CÓDIGO EN ENSAMBLADOR

Los archivos conecta_K_buscar_alineamiento_arm.s y conecta_K_hay_linea_arm_c.s han sido traducidos de las funciones buscar_alineamiento y hay_linea, manteniendo la misma estructura y variables que el código en C. Sin embargo, se han identificado oportunidades de optimización.

En el archivo conecta_K_buscar_alineamiento_arm_opt.s, hemos implementado una versión optimizada del código original de *conecta_K_buscar_alineamiento_arm.s*. La mejora principal es la eliminación de la recursividad, la cual ha sido reemplazada por una búsqueda iterativa (bucle) para buscar el posible alineamiento de la ficha. Se ha introducido una variable local denominada "long_linea," y se ha configurado un bucle que se ejecutará mientras las filas y columnas sigan siendo válidas, y siempre que la celda no esté vacía, y el color coincida con el parámetro pasado. El comportamiento dentro del bucle es consistente con la versión recursiva anterior.

Por otro lado, en el archivo conecta_K_hay_linea_arm_arm.s, hemos implementado la versión optimizada de conecta_K_hay_linea_arm_c.s. En este caso las llamadas a la función buscar alineamiento arm son a la función optimizada. Debido a que no hay recursividad en esta función resulta bastante más óptima. Estas optimizaciones están diseñadas para mejorar el rendimiento y la eficiencia del código de *conecta_K* sin afectar a la funcionalidad del juego.

DESCRIPCIÓN DE PROBLEMAS ENCONTRADOS Y SUS SOLUCIONES

Durante la práctica, nos enfrentamos a diversos problemas que requirieron un análisis exhaustivo. Para depurarlos e identificar la causa y la ubicación de estos errores utilizamos una herramienta de depuración que nos permitió seguir el código instrucción por instrucción.

En primer lugar, antes de empezar a implementar las funciones en ARM, miramos el código generado por el compilador de estas funciones para estudiar su comportamiento.

Nuestro primer problema fue el entender el funcionamiento de la cuadrícula y como indexar las posiciones de la misma para comprobar la celda y el color en la función de *hay_alineamineto_arm*.

Una vez entendido el funcionamiento de la cuadrícula, encontramos otro problema a la hora de implementar las funciones, que sería en este caso la pila , la cual tuvimos que estudiar detenidamente, para conseguir que su uso fuera correcto.

Por ejemplo en la función *buscar_alineamiento* en un principio apilamos todos los registros que se usaban antes de la llamada recursiva y al salir de esta solo hacía un *add sp,sp #8*; esto fue solucionado primeramente poniendo un *add sp,sp #20* para liberar ese espacio de la pila y apuntar al lr para posteriormente hacer el bl.

Posteriormente vimos que no era necesario apilar todos los registros empleados en la función antes de la llamada recursiva ya que los únicos valores que necesitábamos para la siguiente llamada eran los deltas, esto se soluciono apilando solo los deltas.

En la función hay_linea tuvimos un problema a la hora de implementar que delta[i]=-delta[i]. Primero pensamos en negar todos los bits y sumarle 1 como si fuera en complemento a 2, finalmente caímos en la cuenta de la instrucción rsb Reverse Subtract, la cual nos permitía hacer la resta de #0 - delta[i] lo que nos daba el resultado deseado de -delta[i] y además simplificaba bastante el número de instrucciones.

En resumen, la mayoría de los problemas encontrados durante la práctica fueron de comprensión de la cuadrícula, la cual una vez entendida las funciones fueron mera traducción.

El resto de errores fueron a causa del marco de la pila y la gestión de frame pointer para apuntar a los valores determinados.

CONCLUSIONES

Por concluir comentar que hemos conseguido en esta práctica entender de mejor manera el marco de la pila y el paso de parámetro por registro.

También nos ha servido para aprender a traducir funciones de c a arm, y gestionar las posibles optimizaciones del código eliminando recursividad.

Somos consciente de que el código de *hay_linea* se podría optimizar más pero creímos que al realizar las llamadas a la función optimizada de *alineamiento_arm*, se eliminaba la recursividad y por lo tanto la optimización sería notable.

También nos gustaría comentar que gracias al uso de las optimizaciones del compilador nos hemos dado cuenta de cómo de buenas han sido las optimizaciones realizadas con respecto al compilador en -O3 -Otime

Finalmente estamos satisfechos de los resultados obtenidos y del tiempo empleado en la práctica, además de lo beneficiosa que ha resultado para empezar a dominar la pila y las nuevas funcionalidades de arm empleadas en esta práctica.

ANEXO

TABLAS DE MEDIDAS DE RENDIMIENTO

Tiempo (en us)

conecta_K_hay_linea_c_c

	Tiempo
-O0	204.583
-O1	183.834
-O2	166.999
-O3	164.999
-O3 -Otime	95.999

conecta_K_hay_linea_arm_c

	Tiempo
-O0	243.167
-O1	224.000
-O2	204.001
-O3	204.001
-O3 -Otime	146.084

conecta_K_hay_linea_c_arm

Tiempo	Recursivo	Optimizado
-O0	151.834	121.833
-O1	154.417	113.667
-O2	154.417	112.500
-O3	117.886	110.667
-O3 -Otime	117.886	113.000

conecta_K_hay_linea_arm_arm

Tiempo	Recursivo	Optimizado
-O0	144.834	139.000
-O1	144.834	139.000
-O2	144.834	139.000
-O3	144.834	139.000
-O3 -Otime	144.834	139.000

Tamaño del código (en bytes)

conecta_K_hay_linea_c_c

	Tiempo
-O0	2376
-O1	2040
-O2	1828
-O3	1820
-O3 -Otime	2180

conecta_K_hay_linea_arm_c

	Tiempo
-O0	2248
-O1	1964
-O2	1776
-O3	1.780
-O3 -Otime	2.008

conecta_K_hay_linea_c_arm

Tiempo	**Recursivo**	**Optimizado**
-O0	2304	2172
-O1	2176	1896
-O2	2088	1756
-O3	1760	1720
-O3 -Otime	1816	2172

conecta_K_hay_linea_arm_arm

Tiempo	**Recursivo**	**Optimizado**
-O0	2128	2116
-O1	1996	1848
-O2	1916	1704
-O3	1720	1708
-O3 -Otime	1776	1964

CÓDIGO

conecta_K_buscar_alineamiento_arm.s

```
        AREA codigo, CODE, READONLY
        PRESERVE8
        EXPORT conecta_K_buscar_alineamiento_arm

conecta_K_buscar_alineamiento_arm
         mov ip, sp
         stmdb r13!, {r4-r8, fp, lr}
         sub fp, ip, #4
         ; comprobamos si la fila es válida
         cmp r1, #0
         blt fin ; si fila < 0
         cmp r1, #7
         bge fin ; si fila >= 7
         ; comprobamos si la columna es válida
         cmp r2, #0
         blt fin ; si columna < 0
         cmp r2, #7
         bge fin ; si columna >= 7
         ; si está la celda, obtenemos su valor
         mov r8,#0; inicializamos r8 a 0 (contador)
bucle
         cmp r8,#6
         beq fin
         add r6,r1,r1,LSL #1
         add r7,r0,r6,LSL #1
         ldrb r7,[r7,r8]
         add r8,r8,#1 ;cont++
         ; comprobamos si está la celda
         cmp r7,r2
         bne bucle
         sub r8,r8,#1 ; cont-- para recuperar el valor correcto
         add r6,r1,r1,LSL #1
          add r7,r0,#0x0000002A ; buscar en la parte de abajo del tablero (donde se indexan
los colores)
         add r7,r7,r6,LSL #1
         ldrb r7,[r7,r8] ; valor del color
         and r7,r7,#0x03
         cmp r7,r3 ; comparamos el color con el parámetro
         bne fin
         ldrsb r4, [fp, #4] ; r4 - deltas_fila[i]
         add r1, r1, r4 ; r1 - fila + deltas_fila[i]
         ldrsb r5, [fp, #8] ; r5 - deltas_columna[i]
         add r2, r2, r5 ; r2 - columna + deltas_columna[i]
         stmdb r13!, {r4-r5} ; se apilan los deltas para la llamada recursiva
         bl conecta_K_buscar_alineamiento_arm
         add sp, sp, #8
         add r0, r0, #1 ; r0 - r0 + 1 (return 1 + resultado función)
         ldmia r13!, {r4-r8, fp, lr}
         bx r14
fin
         mov r0, #0 ; r0 - 0 (return 0)
         ldmia r13!, {r4-r8, fp, lr}
         bx r14
        END
```

conecta_K_buscar_alineamiento_arm_opt.s

```
	AREA codigo, CODE, READONLY
	PRESERVE8
	EXPORT conecta_K_buscar_alineamiento_arm_opt
conecta_K_buscar_alineamiento_arm_opt
	 mov ip, sp
	 stmdb r13!, {r4-r9, fp, lr}
	 sub fp, ip, #4
	 ; comprobamos si la fila es válida
		ldrsb r4, [fp, #4] ; r4  deltas_fila[i]
		ldrsb r5, [fp, #8] ; r5  deltas_columna[i]
		mov r9, #0 ; r9  long_linea = 0
bucle_ini
		mov r8,#0; inicializamos r8 a 0 (contador)
		cmp r1, #0
	 blt fin ; si fila < 0
	 cmp r1, #7
	 bge fin ; si fila >= 7
	 ; comprobamos si la columna es válida
	 cmp r2, #0
	 blt fin ; si columna < 0
	 cmp r2, #7
	 bge fin ; si columna >= 7
bucle
	 ; si está la celda, obtenemos su valor
	 cmp r8,#6
	 beq fin
	 add r6,r1,r1,LSL #1
	 add r7,r0,r6,LSL #1
	 ldrb r7,[r7,r8]
	 add r8,r8,#1 ;cont++
	 ; comprobamos si está la celda
	 cmp r7,r2
	 bne bucle
	 sub r8,r8,#1 ; cont-- para recuperar el valor correcto
	 add r6,r1,r1,LSL #1
	  add r7,r0,#0x0000002A ; buscar en la parte de abajo del tablero (dnde se indexan
los colores)
	 add r7,r7,r6,LSL #1
	 ldrb r7,[r7,r8] ; valor del color
	 and r7,r7,#0x03
	 cmp r7,r3 ; comparamos el color con el parámetro
	 bne fin
		add r9,r9,#1 ; long_linea++
	 add r1, r1, r4 ; r1 - fila + deltas_fila[i]
	 add r2, r2, r5 ; r2 - columna + deltas_columna[i]
		b bucle_ini
fin
	 mov r0, r9 ; r0 - 0 (return 0)
	 ldmia r13!, {r4-r9, fp, lr}
	 bx r14
		END
```

conecta_K_hay_linea_arm_c.s

```
              AREA datos, DATA, READWRITE
dfila  DCB 0x00, 0xFF, 0xFF, 0x01
dcolum DCB 0xFF, 0x00, 0xFF, 0xFF
ksize   EQU 0x4
              AREA codigo, CODE, READONLY
              PRESERVE8
              IMPORT conecta_K_buscar_alineamiento_c
              EXPORT conecta_K_hay_linea_arm_c
conecta_K_hay_linea_arm_c
    mov ip, sp
    stmdb r13!, {r4-r10, fp, lr}
    sub fp, ip, #4
    ; copia de parámetros
    mov r4, r0 ; r4 cuadricula
    mov r5, r1 ; r5 fila
    mov r6, r2 ; r6 columna
    mov r7, r3 ; r7 color
    mov r8, #0 ; r8 i = 0
    mov r9, #0 ; r9 linea = false
    mov r10, #0 ; r10 long_linea = 0
bucle
    cmp r8, #4 ; i < numDeltas
    bgt fin
    cmp r9, #1 ; linea == True
    beq fin
    ldr r0, =dfila ; r0 dfila
    ldrsb r1, [r0, r8] ; r1 dfila[i]
    ldr r0, =dcolum ; r1 dcolum
    ldrsb r2, [r0, r8] ; r2 dcolum[i]
    stmdb r13!,{r1-r2} ; apilamos dfila[i] y dcolum[i] para buscar alineamiento
    mov r0, r4 ; r0 cuadrícula
    mov r1, r5 ; r1 fila
    mov r2, r6 ; r2 columna
    mov r3, r7 ; r3 color
    bl conecta_K_buscar_alineamiento_c
    ldmia r13!,{r1-r2} ; desapilamos dfila[i] y dcolum[i]
    mov r10, r0 ; long_linea = long_linea + long_linea_actual
    cmp r10, #ksize ; long_linea == K_size
    movge r9, #1 ; linea = true
    bge fin
    rsb r1 ,r1, #0 ; dfila[i] = -dfila[i]
    rsb r2 ,r2, #0 ; dcolum[i] = -dcolum[i]
    stmdb r13!,{r1-r2} ; apilamos -dfila[i] y -dcolum[i] para buscar alineamiento
    mov r0, r4 ; r0 cuadricula
       add r1, r5, r1 ; r1 fila + (-deltas_fila[i])
    add r2, r6, r2 ; r2 columna + (-deltas_columna[i])
    mov r3, r7 ; r3 color
       bl conecta_K_buscar_alineamiento_c ; llamada con la inversa
    ldmia r13!,{r1-r2} ; desapilamos dfila[i] y dcolum[i]
    add r10, r10, r0 ; long_linea = long_linea + long_linea_actual
    cmp r10, #4 ; long_linea == K_size
    movge r9, #1 ; linea = true
    add r8, r8, #1 ; i = i + 1
    b bucle
fin
    mov r0, r9 ; r0 linea
    ldmia r13!, {r4-r10, fp, lr}
    bx r14
END
```

conecta_K_hay_linea_arm_arm.s

```
              AREA datos, DATA, READWRITE
dfila  DCB 0x00, 0xFF, 0xFF, 0x01
dcolum DCB 0xFF, 0x00, 0xFF, 0xFF
ksize   EQU 0x4
              AREA codigo, CODE, READONLY
              PRESERVE8
              IMPORT conecta_K_buscar_alineamiento_arm_opt
              EXPORT conecta_K_hay_linea_arm_arm
conecta_K_hay_linea_arm_arm
    mov ip, sp
    stmdb r13!, {r4-r10, fp, lr}
    sub fp, ip, #4
    ; copia de parámetros
    mov r4, r0 ; r4 cuadricula
    mov r5, r1 ; r5 fila
    mov r6, r2 ; r6 columna
    mov r7, r3 ; r7 color
    mov r8, #0 ; r8 i = 0
    mov r9, #0 ; r9 linea = false
    mov r10, #0 ; r10 long_linea = 0
bucle
    cmp r8, #4 ; i < numDeltas
    bgt fin
    cmp r9, #1 ; linea == true
    beq fin
    ldr r0, =dfila ; r0 dfila
    ldrsb r1, [r0, r8] ; r1 dfila[i]
    ldr r0, =dcolum ; r1 dcolum
    ldrsb r2, [r0, r8] ; r2 dcolum[i]
    stmdb r13!,{r1-r2} ; apilamos dfila[i] y dcolum[i] para buscar alineamiento
    mov r0, r4 ; r0 cuadricula
    mov r1, r5 ; r1 fila
    mov r2, r6 ; r2 columna
    mov r3, r7 ; r3 color
    bl conecta_K_buscar_alineamiento_arm_opt
    ldmia r13!,{r1-r2} ; desapilamos dfila[i] y dcolum[i]
    mov r10, r0 ; long_linea = long_linea + long_linea_actual
    cmp r10, #ksize ; long_linea == K_size
    movge r9, #1 ; linea = true
    bge fin
    rsb r1 ,r1, #0 ; dfila[i] = -dfila[i]
    rsb r2 ,r2, #0 ; dcolum[i] = -dcolum[i]
    stmdb r13!,{r1-r2} ; apilamos -dfila[i] y -dcolum[i] para buscar alineamiento
    mov r0, r4 ; r0 cuadricula
       add r1, r5, r1 ; r1 fila + (-deltas_fila[i])
    add r2, r6, r2 ; r2 columna + (-deltas_columna[i])
    mov r3, r7 ;  r3 color
       bl conecta_K_buscar_alineamiento_arm_opt ; llamada con la inversa
    ldmia r13!,{r1-r2} ; desapilamos dfila[i] y dcolum[i]
    add r10, r10, r0 ; long_linea = long_linea + long_linea_actual
    cmp r10, #4 ; long_linea == K_size
    movge r9, #1 ; linea = true
    add r8, r8, #1 ; i = i + 1
    b bucle
fin
    mov r0, r9 ; r0  linea
    ldmia r13!, {r4-r10, fp, lr}
    bx r14
       END
```