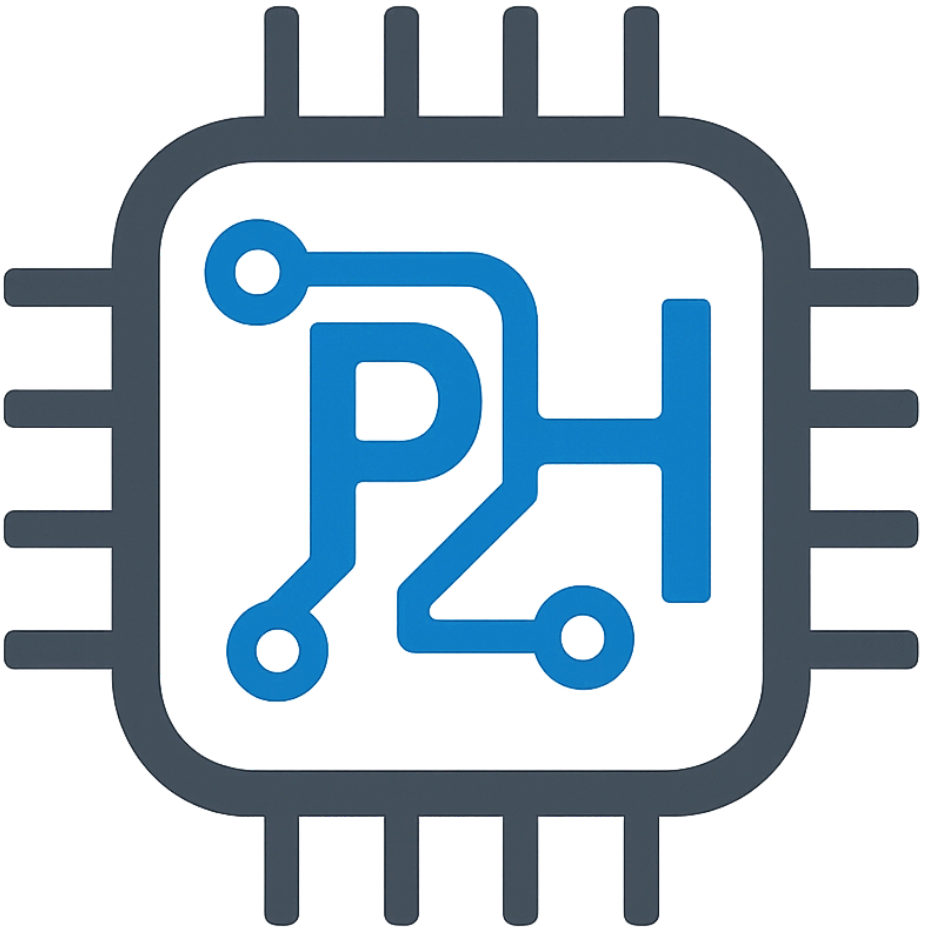
Memoria Proyecto 1 - Proyecto Hardware

Desarrollo de código para el procesador ARM



Autores:

Fernando Pastor Peralta (897113)

Guillermo Ledesma Uche (896594)

Fecha de entrega: 14/10/2025

Grupo: MartesMañana K

Ingeniería Informática - Universidad de Zaragoza, Curso 2025-2026

Índice

Resumen ejecutivo: 3
Problema y profiling inicial: 4
Descripción de las optimizaciones realizadas al código ensamblador: 6
ARM: 6
Thumb: 6
Marco de pruebas 7
Resultados de la comparación entre distintas versiones 8
Descripción de los problemas encontrados en la realización de la práctica y sus soluciones: 9
Conclusiones: 9
Anexo: 10

Resumen ejecutivo

El proyecto tiene como objetivo la evaluación y optimización del rendimiento del microcontrolador LPC2105, basado en un núcleo ARM7TDMI y perteneciente a la arquitectura ARMv4T. Para ello, se ha implementado y analizado una operación convolución 3x3 sobre imágenes de tamaño cualquiera en escala de grises (con valores de píxeles entre 0 y 255); partiendo de un código base en C, se han diseñado versiones equivalentes y optimizadas en ensamblador, tanto en ARM como en Thumb. Para el desarrollo de las versiones en ensamblador se ha respetado el estándar de llamadas ATPCS y hemos verificado su correcto funcionamiento mediante un surtido de pruebas analizado en el entorno Keil µVision.

Hemos seguido una metodología progresiva: para empezar, se comprendió el código en C, y se investigó cómo estaba estructurada la memoria y cómo estaba construido el marco de pila, después, se analizó el rendimiento temporal y espacial mediante la herramienta de Analyze Performance (Profiling) de Keil, finalmente, y en base a los resultados obtenidos, se realizaron versiones optimizadas en ensamblador, tanto orientadas en reducir el tiempo de ejecución (ARM), como en minimizar el tamaño del programa en memoria (Thumb).

La equivalencia funcional entre todas las versiones se comprobó mediante la función conv3x3_verificar, aplicando diferentes juegos de pruebas sobre matrices de distintas dimensiones y kernels representativos (detección de bordes, identidad, etc.).

Los resultados muestran una mejora significativa del rendimiento respecto a la versión original en C compilada normalmente en -O0, y es que tras calcular el speedup, vemos que nuestro código ARM acaba su ejecución aproximadamente un 30% más rápido que el código ensamblador generado por el compilador, mientras que nuestro código en Thumb tiene aproximadamente un 21% menos de bytes que el código original. Además, se han comprobado los distintos niveles de optimización del compilador (O0-O3 y Otime) y hemos observado que los resultados obtenidos por nuestro ensamblador son superiores en algunas optimizaciones y comparable en muchas otras (aunque es visible que hay margen de mejora manual).

En conclusión, la práctica nos permite entender el funcionamiento y perfeccionar el manejo del repertorio de instrucciones del LPC2105, tanto en ARM como en Thumb, además de mostrarnos la importancia de la programación manual en ensamblador para la mejora temporal y espacial del código, y aportar nuevos conocimientos sobre estrategias de optimización para aprovechar al máximo los recursos del procesador.

Problema y profiling inicial

Trabajaremos con el microcontrolador LPC2105, basado en un núcleo ARM7TDMI y perteneciente a la arquitectura ARMv4T, para la realización de optimizaciones en la función conv3x3_operar_u8s8_C que realiza convoluciones sobre imágenes de cualquier tamaño en escala de grises.

Al comienzo de la realización de la práctica, se comenzó con el análisis y depuración del código suministrado en C, entendiendo su funcionamiento y las técnicas utilizadas por el compilador. De este primer análisis, podemos concluir que el código original de la función ocupa 452 bytes de memoria, y tras el primer profiling, vemos que tarda en ejecutarse 563,250us, siendo conv3x3_kernel_u8s8_C la función que más tiempo ocupa con 432,333 us tras 36 llamadas.
Accediendo al código en ARM del compilador, visualizamos el uso de la pila y el reparto de código en el espacio de memoria:

- **Mapa de memoria:**

La memoria disponible en el entorno se puede dividir en tres áreas principales: código, variables y pila.

El mapa comienza con toda la sección de código de todos los ficheros compilados, como es lógico este área es sólo de lectura. Por el contrario, en la zona de pila y variables nos encontramos con que son de lectura y escritura, salvo las constantes como, la imagen o el kernel, que son sólo de lectura.

Pila	@pila	0x400005B4
Variables/datos	@out	0x40000000
	@kernel	0x00000B90
	@imagen	0x00000B50
Código	@main	0x00000A44
	@verificar	0x000008D8
	@operar_C	0x00000830
	@operar_kernel_C	0x00000718
	@operar_THB	0x000004D8
	@operar_ARM	0x0000031C
	@operar_ARM_C	0x00000104

- **Pila:**

Analizando el marco de pila del compilador y el usado en la asignatura Arquitectura y Organización de Computadores 1, nos encontramos con una serie de diferencias. En AOC1, todos los parámetros con los que se llamaba a una función se guardaban en la pila, haciendo uso del fp para recorrerla y obtener de ella sus respectivos valores. En cambio con el protocolo ATPCS y el del compilador, los cuatro primeros registros (r0-r3) almacenan directamente los parámetros (si la función requiere de más de cuatro se empezarían a almacenar en la pila), ya que el contenido de estos registros se preservan entre llamadas.

Por otro lado, almacenar el resultado de la función también varía entre las estructuras. En AOC1 este se guardaba en la pila, mientras que en el compilador y el ATPCS, se usa el registro r0 para pasar el resultado, ya que su valor se preserva una vez se salta a la dirección de retorno.

En el ATPCS empleado, se asume una estructura *"full-descending*"; es decir, que al apilar elementos la pila crece de direcciones altas a bajas y la dirección de la cima (sp) decrece.

COMPILADOR

Variables locales
Registros usados (r4-r11)
LR(r14)
Parámetros

ATPCS

Variables locales
Registros usados (r4-r10)
FP(r11)
LR(r14)
Parámetros

AOC1

Variables temporales
Registros usados
Variables locales
FP
LR
Parámetros
Resultados

Descripción de las optimizaciones realizadas al código ensamblador

ARM:

1. Hemos realizado una implementación in-line de las funciones a las que se llama dentro de la función conv3x3_operar_u8s8, tanto conv3x3_kernel_u8s8, como poner_bordes_a_cero y sat_u8. Esto reduce enormemente la cantidad de saltos que realiza el procesador, además de disminuir la cantidad de veces que hay que acceder a memoria para realizar el prólogo y epílogo de cada una de las funciones. También reduce significativamente la cantidad de cambios en registros, ya que la información dentro de ellos debe estar bien estructurada sin necesidad de estar cargando constantemente información de la pila varias veces.

2. Hemos reducido el número de instrucciones totales respecto a la versión ARM_C, usando instrucciones específicas de ARM para algunas tareas que podrían hacerse normalmente con una cantidad mayor de instrucciones, por ejemplo, usando el postindexado en registros o para evitar hacer multiplicaciones (MUL) y luego sumas en un mismo registro (ADD) usamos la instrucción MLA que lo hace directamente. Por otra parte, evitamos bloques de instrucciones condicionales con instrucciones especiales que miran las flags y dependiendo de la condición de la instrucción se ejecuta o no (ADDNE, por ejemplo).

3. Se hace un uso eficiente de los registros, manteniendo registros dedicados (como img/cols/k/out/i/j en r7, r4, r8, r11, r6 y r1, respectivamente) y usando la pila solo para norm_shift y salvar j, reduciendo accesos de memoria en la función.

4. A la hora de hacer el cálculo de acc (donde se guarda la multiplicación de las matrices) evitamos hacer un bucle para los 9 píxeles y ponemos los cálculos directamente, evitando así que el procesador realice saltos innecesarios.

Thumb:

1. Realizar el cambio al repertorio de instrucciones Thumb supone que tenemos instrucciones de 16 bits, frente a las de 32 bits que tiene ARM, por lo que hacer el programa lo más pequeño posible según las instrucciones que posibilita Thumb, haciéndolo con un número similar o menor de instrucciones que respecto al código en ARM, ya supone una gran mejoría respecto al tamaño de memoria.

2. De nuevo, hemos optado por una implementación in-line de la función, con el mismo objetivo de reducir instrucciones: evitamos prólogos, epílogos y creaciones del contexto necesarios en registros cada vez que se entra a una función. Además, esta implementación nos obliga a hacer un uso especialmente eficiente de los registros

(aún más en Thumb). Además evitamos la instrucción BL, que es la única de 32 bits que tiene Thumb.

3. En la versión Thumb se ha limitado el uso a los registros bajos (r0–r7) y a inmediatos pequeños, ya que el uso de registros altos (r8–r12) o constantes grandes en ARMv4T requiere instrucciones adicionales o literales en memoria, lo que aumenta el tamaño del código.

4. Por supuesto, todos los cambios descritos en la versión ARM que reducen instrucciones se implementan de la misma forma aquí (mientras sea posible).

Marco de pruebas

Para comprobar el funcionamiento del código implementado con todas sus versiones, se han realizado una serie de pruebas. Haciendo uso de la función “verificar” ya dada en el código inicial de la práctica, se ha usado de forma automática para hacer más eficaz y eficiente la verificación. El primer paso era confirmar que nuestro código funcionase para el ejemplo propuesto inicialmente, una matriz 8x8 con los bordes ya a cero a la que se le aplica uno de los kernel de detección de bordes vistos en la teoría. Una vez la función verificar devolvía el resultado que confirmaba que todas las matrices resultado eran iguales para todas las versiones, se continuó con la realización de las otras dos pruebas.

La primera de ellas consistía en una matriz del mismo tamaño que la anterior, pero en cada esquina se encuentra el valor “0x70”. Esto es para comprobar que todas las implementaciones de la función “poner bordes a cero” funcionaban correctamente. Además se ha aplicado un kernel distinto de detección de bordes también visto en las transparencias, para verificar si con otros valores se realizan las multiplicaciones correctamente. Todo esto se ha hecho declarando nuevas variables para las matrices “imagen”, “resultado” y “kernel”, y llamando de nuevo a la función “verificar” con estos nuevos parámetros.

Por último se ha propuesto cambiar las dimensiones de la matriz a 5x4, para que tenga otro tamaño y proporción distinta (no cuadrada), aplicando el kernel identidad que devuelve como resultado la imagen sin ninguna modificación. Empleando el mismo método que en el caso anterior (pasando las dimensiones a la función manualmente ya que las constantes globales de filas y columnas no se pueden cambiar), se ha comprobado que para dimensiones distintas a 8x8 todo funciona como debería.

Importante: En este último caso, se pasaba a la función verificar la matriz resultado 8x8 en vez de la 5x4, lo que generaba un “*warning*” en la compilación y al ejecutar el código hacía mal los cálculos. Esto se soluciona cambiando el parámetro a la variable ya creada “resultado3” en el código main, aunque este error fue detectado una vez realizada la entrega del código.

Resultados de la comparación entre distintas versiones

A continuación se desglosan los dos principales análisis del código: temporal y espacial.

- **Análisis temporal:**

Para este caso se ha utilizado la herramienta del Keil llamada Analyze Performance (Profiling), que proporciona el coste de tiempo de todos los módulos conforme se va ejecutando el código. Para comprobar si se vencía al compilador o no, se ha ejecutado con distintos flags (O0, O1, O2, O3 y Otime), que realizan una serie de optimizaciones al código para mejorar su rendimiento, estas son:

- **O0:** sin optimizaciones.
- **O1:** reduce el coste de ejecución sin incrementar mucho el tamaño o el tiempo de compilación.
- **O2:** ofrece un buen equilibrio entre el rendimiento del código y el tiempo de compilación, y suele ser el nivel recomendado, emplea optimizaciones más agresivas pero seguras.
- **O3:** aplica optimizaciones más agresivas para obtener el máximo rendimiento. Puede aumentar el tamaño del código y el tiempo de compilación, y a veces puede causar problemas con la depuración.
- **Otime:** esta opción le indica al compilador que priorice la mejora de tiempo antes que la de tamaño.

	operar_C	operar_ARM_C	operar_ARM	operar_THB
-O0	**563,250us**	553,250us	434,167us	567us
-O1	**504,916us**	553,250us	434,167us	567us
-O2 + Otime	**369,833us**	553,250us	434,167us	567us
-O3 + Otime	**369,833us**	553,250us	434,167us	567us

- **Tamaño código:**

	operar_C	operar_ARM_C	operar_ARM	operar_THB
-O0	**436 bytes**	536 bytes	444 bytes	344 bytes
-O1	**452 bytes**	536 bytes	444 bytes	344 bytes
-O2	**412 bytes**	536 bytes	444 bytes	344 bytes
-O3	**412 bytes**	536 bytes	444 bytes	344 bytes

Descripción de los problemas encontrados en la realización de la práctica y sus soluciones

- Adaptación a las nuevas instrucciones en ARM que no conocíamos.

- Dificultades iniciales para encontrar optimizaciones más allá de usar ciertas instrucciones ARM.

- Comprensión del nuevo repertorio de instrucciones Thumb y el uso limitado de registros en dicha arquitectura, ya que se quería evitar usar registros altos que requieren de instrucciones más costosas.

Conclusiones

Como síntesis total de nuestro trabajo, nuestros resultados muestran una mejora significativa respecto al código compilado en C en su versión de compilación más básica:

- En ARM, tras el cálculo del speedup, vemos que nuestro código va un 30% más rápido que el código generado por el compilador: (563,25/434,167-1)*100 ≈ 30%

- En Thumb, vemos que el código tiene un 21% menos de bytes que el código generado por el compilador: (1-344/436)*100 ≈ 21%

Por último, hemos aprendido a dominar el repertorio de instrucciones del LPC2105, además de técnicas de optimización para el mismo y hemos comprendido la importancia de la programación manual en ensamblador para mejorar la optimización de nuestro código.

Anexo

- **Código ARM_C:**

```
AREA    |.text|, CODE, READONLY, ALIGN=2
        EXPORT  conv3x3_operar_u8s8_ARM_C
        EXPORT  poner_bordes_a_cero_ARM_C
                EXPORT  conv3x3_kernel_u8s8_ARM_C
        PRESERVE8
        ARM


;Función conv3x3_kernel_u8s8_ARM_C:
;(a pesar de no ser pedido lo ponemos ya que lo teníamos ya hecho y no modifica
;nada temporalmente)

;- Cómo funciona: Calcula la convolución 3x3 en una posición (i,j) de una imagen
;uint8 con kernel int8, aplica shift de normalización y arreglo entre 0 y 255.

;- Parámetros: img (r0), cols (r1), k (r2), norm_shift (r3), i ([sp,#32]), j ([sp,#36]).

;- Registros: r4=img, r5=cols, r6=k, r7=norm shift, r8=p (puntero inicial), r10=p
;(puntero dinámico), r11=j ajustado, r2=acc, r0=resultado final, r1=auxiliar para
;multiplicaciones y bias.

conv3x3_kernel_u8s8_ARM_C PROC
                ;r0 = img, r1 = cols, r2 = k, r3 = norm_shift
                STMDB sp!, {r4-r8, r10, r11, lr}        ;prólogo

                MOV r4, r0                              ;r4 = img
                MOV r5, r1                              ;r5 = cols
                MOV r6, r2                              ;r6 = k
                MOV r7, r3                              ;r7 = norm_shift

                LDR r10, [sp, #32]              ;r10 = i
                LDR r11, [sp, #36]              ;r11 = j

                SUB r10, r10, #1                ;r10 = i - 1
                SUB r11, r11, #1                ;r11 = j - 1
                MLA r10, r5, r10, r11   ;r10 = (i - 1) * cols + (j - 1)
                ADD r10, r4, r10                ;r10 = p = img + (i - 1) * cols + (j - 1)
                MOV r8, r10                             ;r8 = p

                                                                ;r2 = acc
                LDRSB r0, [r6], #1              ;r0 = k[0], r6 = r6 + 1
                LDRB r1, [r10], #1              ;r1 = p[0], r10 = r10 +1
```

```
                MUL r2, r0, r1          ;r2 = k[0]*p[0]
                        LDRSB r0, [r6], #1      ;r0 = k[1], r6 = r6 + 1
                LDRB r1, [r10], #1      ;r1 = p[1], r10 = r10 + 1
                MLA r2, r0, r1, r2      ;r2=k[1]*p[1]+k[0]*p[0]
                LDRSB r0, [r6], #1      ;r0 = k[2], r6 = r6 + 1
                LDRB r1, [r10], #1      ;r1 = p[2], r10 = r10 + 1
                MLA r2, r0, r1, r2      ;r2=k[2]*p[2]+k[1]*p[1]+k[0]*p[0]
                LDRSB r0, [r6], #1      ;r0 = k[3], r6 = r6 + 1
                ADD r10, r8, r5                 ;r10 = p + cols
                LDRB r1, [r10], #1      ;r1 = p[cols]
                MLA r2, r0, r1, r2      ;r2=k[3]*p[cols]+k[2]*p[2]+k[1]*p[1]+k[0]*p[0]
                LDRSB r0, [r6], #1      ;r0 = k[4], r6 = r6 + 1
                LDRB r1, [r10], #1      ;r1 = p[cols+1]
                MLA r2, r0, r1, r2      ;r2=k[4]*p[cols+1]+k[3]*p[cols]+...
                LDRSB r0, [r6], #1      ;r0 = k[5], r6 = r6 + 1
                LDRB r1, [r10], #1      ;r1 = p[cols+2]
                MLA r2, r0, r1, r2      ;r2=k[5]*p[cols+2]+k[4]*p[cols+1]+*...
                ADD r10, r8, r5, LSL #1 ;r10=p+2*cols
                LDRSB r0, [r6], #1      ;r0 = k[6], r6 = r6 + 1
                LDRB r1, [r10], #1      ;r1 = p[2*cols]
                MLA r2, r0, r1, r2      ;r2=k[6]*p[2*cols]+k[5]*p[cols+2]+...
                LDRSB r0, [r6], #1      ;r0 = k[7], r6 = r6 + 1
                LDRB r1, [r10], #1      ;r1 = p[2*cols+1]
                MLA r2, r0, r1, r2      ;r2=k[7]*p[2*cols+1]+k[6]*p[2*cols]+...
                LDRSB r0, [r6], #1      ;r0 = k[8], r6 = r6 + 1
                LDRB r1, [r10], #1      ;r1 = p[2*cols+2]
                MLA r2, r0, r1, r2      ;r2=k[8]*p[2*cols+2]+k[7]*p[2*cols+1]+...

                MOV r0, r2                      ;r0 = acc

                CMP r7, #0                      ;si norm_shift <= 0 salta a fin_if
                BLE fin_if                      ;

                SUB r2, r7, #1          ;r2 = norm_shift-1
                MOV r1, #1                      ;r1 = 1
                MOV r1, r1, LSL r2      ;r1 = bias = 1 << (norm_shift-1)

                CMP r0, #0                      ;si acc < 0 salta a else
                BLT else1
if1
                ADD r1, r0, r1          ;r1 = acc + bias
                MOV r0, r1, ASR r7      ;r0 = acc = (acc + bias) << norm_shift
                B fin_if
else1
                SUB r1, r0, r1          ;r1 = acc - bias
                MOV r0, r1, ASR r7      ;r0 = acc = (acc - bias) << norm_shift
fin_if
```

```
                CMP r0, #0                      ;si acc >= 0 salta a if2
                BGE if2                                 ;
                MOV r0, #0                      ;devolvemos 0
                B fin_if2
if2
                CMP r0, #255            ;si acc <= 255 salta al final
                BLE fin_if2                     ;
                MOV r0, #255            ;devolvemos 255
fin_if2


                ;epílogo
                LDMIA r13!, {r4-r8, r10, r11, lr}
                BX lr
        ENDP
```

;Función poner_bordes_a_cero_ARM_C:
;(a pesar de no ser pedido lo ponemos ya que lo teníamos ya hecho y no modifica ;nada temporalmente)

;- Cómo funciona: Establece a cero los bordes (primera/última fila y primera/última ;columna) de una matriz de salida.

;- Parámetros: out (r0), filas (r1), cols (r2).

;- Registros: r4=out, r5=cols, r6=filas, r7=cols-1, r8=r, r11=auxiliar, r0=0, r1=puntero a ;fila 0, r2=puntero fila inferior, r3=contador columnas, r12=filas-1.

```
poner_bordes_a_cero_ARM_C PROC
        STMDB sp!, {r4-r8, r10, r11, lr}  ;prólogo

        MOV r4, r0              ;r4 = out
        MOV r6, r1                 ;r6 = filas
        MOV r5, r2                 ;r5 = cols

        SUB r0, r6, #1                  ;r0 = filas -1
        MUL r11, r0, r5                 ;r11= (filas-1)*cols
        ADD r11, r4, r11          ;r11= @out +(filas-1)*cols

        MOV r1, r4                   ;r1 = @out
        MOV r2, r11               ;r2 = @out +(filas-1)*cols
        MOV r3, r5                ;c = cols
        MOV r0, #0                              ;r0 = 0
primer_for
        STRB r0, [r1], #1         ;MemB[@out] = 0, r1=r1+1 (arriba)
        STRB r0, [r2], #1         ;MemB[@out + r2] = 0, r2=r2+1 (abajo)
        SUBS r3, r3, #1                 ;c--, act. flags
```

```
	BNE primer_for			;si c != 0 salta a primer_for

	CMP r6, #2				;si filas <= 2 salta a fors_fin
	BLE fors_fin			;

	SUB r12, r6, #1			;r12= filas-1
	SUB r7, r5, #1			;r7 = cols-1
	MOV r8, #1				;r = r8 = 1
segundo_for
	MUL r2, r8, r5			;r2 = r*cols
	ADD r1, r4, r2			;r1 = @out + r*cols
	STRB r0, [r1]			;MemB[@out + r*cols] = 0 (izda)
	STRB r0, [r1, r7]		;MemB[@out + r*cols + cols-1] = 0 (dcha)
	ADD r8, r8, #1			;r++
	CMP r8, r12				;si r < filas-1 salta a segundo_for
	BLT segundo_for			;
fors_fin
		;epílogo
	LDMIA sp!, {r4-r8, r10, r11, lr}
	BX lr
	ENDP
```

;Función conv3x3_operar_u8s8_ARM_C:

;- Cómo funciona: Aplica convolución 3x3 completa sobre imagen uint8 con kernel ;int8, pone bordes a cero en salida, cuenta píxeles, nonzero es el resultado.

;- Parámetros: img (r0), filas (r1), cols (r2), k (r3), out ([sp,#32]), norm_shift ([sp,#36]).

;- Registros: r4=cols, r5=filas, r6=i, r7=img, r8=k, r10=nonzero, r11=out, r0=auxiliar ;para cálculos y retorno, r1=j, r2=auxiliar para direcciones, r3=auxiliar.

```
conv3x3_operar_u8s8_ARM_C PROC
	STMDB sp!, {r4-r8, r10, r11, lr}  ;prólogo

	MOV r7, r0				;r7 = img
	MOV r5, r1				;r5 = filas
	MOV r4, r2				;r4 = cols
	MOV r8, r3				;r8 = k
	LDR r11, [sp, #32]		;r11= out

		;poner_bordes_a_cero(out, filas, cols)
	MOV r0, r11				;out
	MOV r1, r5				;filas
	MOV r2, r4				;cols
	BL poner_bordes_a_cero_ARM_C
```

```
        MOV r10, #0                     ;r10 = nonzero = 0
        MOV r6,  #1                     ;r6 = i = 1

for_i
        SUB r0,  r5, #1         ;r0 = filas -1
        CMP r6,  r0                     ;si i >= filas-1 saltar a fin_for
        BGE fin_for                     ;

        MOV r1,  #1                     ;r1 = j = 1

for_j
        SUB r0,  r4, #1         ;r0 = cols -1
        CMP r1,  r0                     ;si j >= cols-1 salta a siguiente_i
        BGE siguiente_i         ;

                ;conv3x3_kernel_u8s8_C(img, cols, k, norm_shift, i, j)
        MOV r0, r7                      ;img
        MOV r2, r8                      ;k
        LDR r3, [sp, #36]       ;norm_shift
        SUB sp, sp, #8          ;dejamos espacios en la pila para 2 parámetros
        STR r6, [sp]            ;i
        STR r1, [sp, #4]        ;j
        MOV r1, r4                      ;cols
        BL conv3x3_kernel_u8s8_ARM_C    ;Devuelve r0 = y
        LDR r1, [sp, #4]        ;recuperamos j en r1
        ADD sp, sp, #8          ; quitamos espacios dedicados a los 2 parámetros

        MUL r2, r6, r4          ;r2 = i*cols
        ADD r2, r2, r1          ;r2 = i*cols + j
        ADD r2, r11, r2         ;r2 = @out + i*cols + j
        STRB r0, [r2]           ;MemB[@out + i*cols + j] = y
        CMP r0, #0                      ;si y != 0 nonzero++
        ADDNE r10, r10, #1      ;

        ADD r1, r1, #1          ;j++
        B for_j

siguiente_i
        ADD r6, r6, #1          ;i++
        B for_i

fin_for
        MOV r0, r10                     ;return nonzero

                ;epílogo
        LDMIA sp!, {r4-r8, r10, r11, lr}
        BX lr
        ENDP
```

```
    END
```

- **Código ARM:**

```
AREA    |.text|, CODE, READONLY, ALIGN=2
                EXPORT  conv3x3_operar_u8s8_ARM
                PRESERVE8
                ARM
```

;Función conv3x3_operar_u8s8_ARM:

;- Cómo funciona: Versión optimizada ARM de convolución 3x3 completa: pone ;bordes a cero inline, calcula kernel inline por posición, cuenta en nonzero.

;- Parámetros: img (r0), filas (r1), cols (r2), k (r3), out ([sp,#32]), norm_shift ([sp,#36]).

;- Registros: r4=cols, r5=filas, r6=cols-1 o i (según la sección), r7=img, r8=k, ;r10=nonzero, r11=out, r12=auxiliar para punteros y límites, r0=auxiliar/acc/return, ;r1=auxiliar/j, r2=auxiliar/acumulador, r3=auxiliar/norm_shift o r.

```
conv3x3_operar_u8s8_ARM PROC
                STMDB   sp!, {r4-r8, r10, r11, lr}

                ;r0=img, r1=filas, r2=cols, r3=k, [sp,#32]=out, [sp,#36]=norm_shift
                MOV r7, r0                      ;r7 = img
                MOV r5, r1                      ;r5 = filas
                MOV r4, r2                      ;r4 = cols
                MOV r8, r3                      ;r8 = k
                LDR r11, [sp, #32]          ;r11 = img
                LDR r3, [sp, #36]           ;r3 = norm_shift

                ;Guarda 2 espacios de pila sp[0]=norm_shift y sp[4]=j
                SUB sp, sp, #8                  ;
                STR r3, [sp]                ;guardamos para luego norm_shift

                ;------------------ poner_bordes_a_cero ------------------
                ;Usamos r0-r3,r6,r12 como temporales.
                ;No tocamos r4=cols, r5=filas, r7=img, r8=k, r10, r11=out.

                MOV r1, r11                     ;r1 = @out
                SUB r0, r5, #1              ;r0 = filas - 1
                MLA r2, r0, r4, r11         ;r2 = @out + (filas - 1)*cols
                MOV r3, r4                      ;c = cols
                MOV r0, #0                      ;r0 = 0

primer_for
```

```
                STRB r0, [r1], #1               ;MemB[@out] = 0, r1=r1+1 (arriba)
                STRB r0, [r2], #1               ;MemB[@out + r2] = 0, r2=r2+1 (abajo)
                SUBS r3, r3, #1                         ;c--, act.flags
                BNE primer_for

                CMP r5, #2                              ;si filas <= 2 salta a fors_fin
                BLE fors_fin                    ;

                SUB r12, r5, #1                         ;r12 = filas - 1
                SUB r6, r4, #1                  ;r6 = cols - 1
                MOV r3, #1                              ;r = r3 = 1

segundo_for
                MLA r1, r3, r4, r11             ;r1 = @out + r*cols
                STRB r0, [r1]                   ;MemB[@out + r*cols] = 0 (izda)
        STRB r0, [r1, r6]           ;MemB[@out + r*cols + cols-1] = 0 (dcha)
                ADD r3, r3, #1                  ;r++
                CMP r3, r12                             ;si r < filas-1 salta a segundo_for
                BLT segundo_for                         ;

fors_fin
        ;---------------- FIN poner_bordes_a_cero ----------------

                MOV r10, #0                             ;nonzero = r10 = 0
                MOV r6, #1                              ;i = r6 = 1

for_i
                SUB r0, r5, #1                  ;r0 = filas - 1
                CMP r6, r0                              ;si i >= filas-1 salta a fin_for
                BGE fin_for                             ;

                MOV r1, #1                              ;j = r1 = 1

for_j
                SUB r0, r4, #1                  ;r0 = cols - 1
                CMP r1, r0                              ;si j >= cols-1 salta a siguiente_i
                BGE siguiente_i                         ;

        ;------------------ conv3x3_kernel_u8s8_C ------------------
                ;Usamos r0-r3,r12 como temporales.
        ;No tocamos r7=img, r4=cols, r8=k, [sp]=norm_shift, r6=i, r1=j

                STR r1, [sp, #4]                ;salvamos j

                SUB r12, r6, #1                         ;r12 = i - 1
                MOV r3, r1                              ;r3 = j
                SUB r1, r3, #1                  ;r1 = j - 1
                MLA r12, r4, r12, r1    ;r12 = (i-1)*cols + (j-1)
```

```
        ADD r12, r7, r12            ;p = r12 = img + (i-1)*cols (j-1)

        MOV r0, r8                  ;k_cambiante = r0

        ;r2 = acc
        ;--- fila 0 ---
        LDRSB r3, [r0], #1          ;r3 = k[0], r0 = r0 + 1
        LDRB r1, [r12], #1          ;r1 = p[0], r12 = r12 +1
        MUL r2, r3, r1              ;r2 = k[0]*p[0]

LDRSB r3, [r0], #1          ;r3 = k[1], r0 = r0 + 1
        LDRB r1, [r12], #1          ;r1 = p[1], r12 = r12 +1
        MLA r2, r3, r1, r2          ;r2 = k[1]*p[1]+k[0]*p[0]


LDRSB r3, [r0], #1          ;r3 = k[2], r0 = r0 + 1
        LDRB r1, [r12], #1          ;r1 = p[2], r12 = r12 +1
        MLA r2, r3, r1, r2          ;r2 = k[2]*p[2]+k[1]*p[1]+k[0]*p[0]

ADD r12, r12, r4        ;p = p + cols
SUB r12, r12, #3        ;p = p + cols - 3 (porque avanzamos 3 en la fila)

;--- fila 1 ---
LDRSB r3, [r0], #1          ;r3 = k[3], r0 = r0 + 1
        LDRB r1, [r12], #1          ;r1 = p[cols], r12 = r12 +1
        MLA r2, r3, r1, r2          ;r2 = k[3]*p[cols]+k[2]*p[2]+k[1]*p[1]+k[0]*p[0]

        LDRSB r3, [r0], #1          ;r3 = k[4], r0 = r0 + 1
        LDRB r1, [r12], #1          ;r1 = p[cols+1], r12 = r12 +1
        MLA r2, r3, r1, r2          ;r2 = k[4]*p[cols+1]+k[3]*p[cols]+...

        LDRSB r3, [r0], #1          ;r3 = k[5], r0 = r0 + 1
        LDRB r1, [r12], #1          ;r1 = p[cols+2], r12 = r12 +1
        MLA r2, r3, r1, r2          ;r2 = k[5]*p[cols+2]+k[4]*p[cols+1]+*...

ADD    r12, r12, r4     ;p = p + cols
SUB    r12, r12, #3     ;p = p + cols - 3 (porque avanzamos 3 en la fila)

;--- fila 2 ---
        LDRSB r3, [r0], #1          ;r3 = k[6], r0 = r0 + 1
        LDRB r1, [r12], #1          ;r1 = p[2*cols], r12 = r12 +1
        MLA r2, r3, r1, r2          ;r2 = k[6]*p[2*cols]+k[5]*p[cols+2]+...

LDRSB r3, [r0], #1          ;r3 = k[7], r0 = r0 + 1
        LDRB r1, [r12], #1          ;r1 = p[2*cols+1], r12 = r12 +1
        MLA r2, r3, r1, r2          ;r2 = k[7]*p[2*cols+1]+k[6]*p[2*cols]+...

LDRSB r3, [r0], #1          ;r3 = k[8], r0 = r0 + 1
```

```
                LDRB r1, [r12], #1          ;r1 = p[2*cols+2], r12 = r12 +1
                MLA r2, r3, r1, r2          ;r2 = k[8]*p[2*cols+2]+k[7]*p[2*cols+1]+...

        MOV r0, r2                          ;r0 = acc
                LDR r3, [sp]                ;r3 = norm_shift

                CMP r3, #0                          ;si norm_shift <=0 salta a fin_if
                BLE fin_if                          ;

                SUB r2, r3, #1              ;r2 = norm_shift - 1
                MOV r1, #1                          ;r1 = 1
                MOV r1, r1, LSL r2          ;bias = r1 = 1 << (norm_shift-1)

                CMP r0, #0                          ;si acc < 0 salta a else1
                BLT else1                           ;

if1
                ADD r1, r0, r1              ;r1 = acc + bias
                MOV r0, r1, ASR r3          ;r0 = acc = (acc + bias) << norm_shift
        B fin_if
else1
        SUB r1, r0, r1                  ;r1 = acc - bias
        MOV r0, r1, ASR r3              ;r0 = acc = (acc - bias) << norm_shift
fin_if
                CMP r0, #0                          ;si acc >= 0 salta a if2
                BGE if2                                     ;
        MOV r0, #0                          ;devolvemos 0
        B fin_if2
if2
        CMP r0, #255                    ;si acc <= 255 salta a fin_if2
        BLE fin_if2                             ;
        MOV r0, #255                    ;devolvemos 255
fin_if2
                ;---------------- FIN conv3x3_kernel_u8s8_C ----------------

        ; restaurar j y escribir salida
        LDR r1, [sp, #4]        ;recuperamos j
                MLA r2, r6, r4, r1          ;r2 = i*cols + j
                ADD r2, r11, r2                     ;r2 = @out + i*cols + j
                STRB r0, [r2]               ;MemB[@out + i*cols + j] = y

        CMP r0, #0                          ;si y != 0 nonzero++
        ADDNE r10, r10, #1              ;

        ADD r1, r1, #1              ;j++
        B for_j

siguiente_i
```

```
        ADD r6, r6, #1              ;i++
        B for_i

fin_for
                MOV r0, r10                             ;return nonzero

                ADD sp, sp, #8                          ;liberamos la zona reservada a par
metros locales

                LDMIA sp!, {r4-r8, r10, r11, lr}
        BX lr
        ENDP

        END
```

- **Código Thumb:**

```
 AREA    |.text|, CODE, READONLY, ALIGN=2
        EXPORT  conv3x3_operar_u8s8_THB
        PRESERVE8

        ARM
```

;Función: conv3x3_operar_u8s8_THB:

;- Cómo funciona: Versión Thumb de convolución 3x3: pone bordes a cero inline, ;calcula kernel inline, cuenta en nonzero, usa pila para variables locales.

;- Parámetros: img (r0), filas (r1), cols (r2), k (r3), out ([sp,#32]), norm_shift ([sp,#36]).

;- Registros: r0=i/acc/retorno/aux, r1=aux/cols-1/bias, r2=j/aux/punteros, r3=aux/k ;dinámico/r (fila), r4=cols, r5=filas, r6=filas-1/aux/k inicial, r7=out o p, pila: ;[sp,#0]=norm_shift, [sp,#4]=j, [sp,#8]=nonzero, [sp,#12]=img, [sp,#16]=k, ;[sp,#20]=out, [sp,#24]=i.

```
conv3x3_operar_u8s8_THB PROC
        STMDB   sp!, {r4-r8, r10, r11, lr}   ; 32 bytes

        ADR     r12, to_thb
        ADD     r12, r12, #1
        BX      r12

        THUMB
to_thb
                ;r0=img, r1=filas, r2=cols, r3=k, [sp,#32]=out, [sp,#36]=norm_shift
                MOVS r5, r1             ;r5 = filas
                MOVS r4, r2             ;r4 = cols
```

```
                LDR r7, [sp, #32]       ;r7 = out
                LDR r1, [sp, #36]       ;r1 = norm_shift

                ;Guarda 7 espacios de pila sp[0]=norm_shift, sp[4]=j, sp[8]=nonzero,
sp[12]=img, sp[16]=k, sp[20]=out y sp[24]=i
                SUB sp, sp, #28                 ;Creamos los 7 espacios en pila
                STR r1, [sp]                    ;sp[0]=norm_shift
                MOVS r1, #0
                STR r1, [sp, #8]                ;sp[8]=nonzero = 0
                STR r0, [sp, #12]               ;sp[12]=img
                STR r3, [sp, #16]               ;sp[16]=k
                STR r7, [sp, #20]               ;sp[20]=out

                ;------------------ poner_bordes_a_cero ------------------
                SUBS r6, r5, #1                 ;r6 = filas-1
                MOVS r2, r6                     ;r2 = filas-1
                MULS r2, r4, r2         ;r2 = (filas-1)*cols
                ADDS r2, r7, r2             ;r2 = @out + (filas-1)*cols
                MOVS r3, r4             ;c = cols
                MOVS r0, #0                     ;r0 = 0

primer_for
                STRB r0, [r7]           ;MemB[@out] = 0 (arriba)
                ADDS r7, r7, #1                 ;r7=r7+1
                STRB r0, [r2]                   ;MemB[r2] = 0 (abajo)
                ADDS r2, r2, #1                 ;r2=r2+1

                SUBS r3, r3, #1                 ;c--, act.flags
                BNE primer_for

                CMP r5, #2                      ;si filas <= 2 salta a fors_fin
                BLE fors_fin                    ;

                MOVS r3, #1             ;r = r3 = 1
                SUBS r1, r4, #1         ;r1 = cols-1

segundo_for
                LDR r7, [sp, #20]       ;r7 = @out (el inicial)
                MOVS r2, r3                     ;r2 = r
                MULS r2, r4, r2             ;r2 = r*cols
                ADDS r7, r7, r2                 ;r7 = @out + r*cols
                STRB r0, [r7]                   ;MemB[@out + r*cols] = 0 (izda)
                STRB r0, [r7, r1]               ;MemB[@out + r*cols + cols-1] = 0 (dcha)
                ADDS r3, r3, #1                 ;r++

                CMP r3, r6                      ;si r < filas-1 salta a segundo_for

                BLT segundo_for                 ;
```

```
fors_fin
                ;---------------- FIN poner_bordes_a_cero ----------------
                ;r1 = cols-1, r4= cols, r5 = filas, r6= filas-1, r0,r2,r3,r7
                MOVS r0, #1                 ;i = r0 = 1

for_i
                SUBS r3, r5, #1         ;r3 = filas-1
                CMP r0, r3                              ;si i >= filas-1 salta a fin_for
                BGE fin_for                             ;

                MOVS r2, #1             ;j = r2 = 1

for_j
                SUBS r3, r4, #1         ;r3 = cols-1
                CMP r2, r3                              ;si j >= cols-1 salta a siguiente_i
                BGE siguiente_i                         ;

;------------------ conv3x3_kernel_u8s8_C ------------------
;r0 = i, r1 = cols-1, r2 = j, r4= cols, r5 = filas, r6= filas-1,r3,r7
                STR r2, [sp, #4]        ;salvamos j
                STR r0, [sp, #24]       ;salvamos i

                ; p = img + (i-1)*cols + (j-1)
                LDR r3, [sp, #12]       ;r3 = img
                SUBS r7, r0, #1         ;r7 = i - 1
                MULS r7, r4, r7         ;r7 = (i-1)*cols
                SUBS r2, r2, #1         ;r2 = j-1
                ADDS r7, r7, r2                         ;r7 = (i-1)*cols + (j-1)
                ADDS r7, r7, r3         ;p = r7 = img + (i-1)*cols + (j-1)

                LDR r3, [sp, #16]       ;k_cambiante = r3
                MOVS r0, #0             ;acc = r0

                MOVS r2, #0

        ;--- fila 0 ---
        LDRSB r6, [r3, r2]      ;r6 = k[0]
        ADDS r3, r3, #1                 ;r3=r3+1
        LDRB r1, [r7]           ;r1 = p[0]
        ADDS r7, r7, #1                 ;r7=r7+1
        MULS r1, r6, r1         ;r1 = k[0]*p[0]
        ADDS r0, r0, r1                 ;r0 = k[0]*p[0]

        LDRSB r6, [r3, r2]      ;r6 = k[1]
        ADDS r3, r3, #1                 ;r3=r3+1
        LDRB r1, [r7]           ;r1 = p[1]
        ADDS r7, r7, #1                 ;r7=r7+1
        MULS r1, r6, r1         ;r1 = k[1]*p[1]
```

```
ADDS r0, r0, r1                 ;r0 = k[0]*p[0]+k[1]*p[1]

LDRSB r6, [r3, r2]     ;r6 = k[2]
ADDS r3, r3, #1                 ;r3=r3+1
LDRB r1, [r7]          ;r1 = p[2]
ADDS r7, r7, #1                 ;r7=r7+1
MULS r1, r6, r1         ;r1 = k[2]*p[2]
ADDS r0, r0, r1                 ;r0 = k[2]*p[2]+k[1]*p[1]+k[0]*p[0]


ADDS    r7, r7, r4              ;p = p + cols
SUBS    r7, r7, #3              ;p = p + cols - 3 (porque avanzamos 3 en la fila)

;--- fila 1 ---
LDRSB r6, [r3, r2]     ;r6 = k[3]
ADDS r3, r3, #1                 ;r3=r3+1
LDRB r1, [r7]          ;r1 = p[cols]
ADDS r7, r7, #1                 ;r7=r7+1
MULS r1, r6, r1         ;r1 = k[3]*p[cols]
ADDS r0, r0, r1                 ;r0 = k[3]*p[cols]+k[2]*p[2]+k[1]*p[1]+k[0]*p[0]

LDRSB r6, [r3, r2]     ;r6 = k[4]
ADDS r3, r3, #1                 ;r3=r3+1
LDRB r1, [r7]          ;r1 = p[cols+1]
ADDS r7, r7, #1                 ;r7=r7+1
MULS r1, r6, r1         ;r1 = k[4]*p[cols+1]
ADDS r0, r0, r1                 ;r0 = k[4]*p[cols+1]+k[3]*p[cols]+...

LDRSB r6, [r3, r2]     ;r6 = k[5]
ADDS r3, r3, #1                 ;r3=r3+1
LDRB r1, [r7]          ;r1 = p[cols+2]
ADDS r7, r7, #1                 ;r7=r7+1
MULS r1, r6, r1         ;r1 = k[5]*p[cols+2]
ADDS r0, r0, r1                 ;r0 = k[5]*p[cols+2]+k[4]*p[cols+1]+*...


ADDS    r7, r7, r4              ;p = p + cols
SUBS    r7, r7, #3              ;p = p + cols - 3 (porque avanzamos 3 en la fila)

;--- fila 2 ---
LDRSB r6, [r3, r2]     ;r6 = k[6]
ADDS r3, r3, #1                 ;r3=r3+1
LDRB r1, [r7]          ;r1 = p[2*cols]
ADDS r7, r7, #1                 ;r7=r7+1
MULS r1, r6, r1         ;r1 = k[6]*p[2*cols]
ADDS r0, r0, r1                 ;r0 = k[6]*p[2*cols]+k[5]*p[cols+2]+...

            LDRSB r6, [r3, r2]      ;r6 = k[7]
```

```
        ADDS r3, r3, #1                 ;r3=r3+1
        LDRB r1, [r7]           ;r1 = p[2*cols+1]
        ADDS r7, r7, #1                 ;r7=r7+1
        MULS r1, r6, r1          ;r1 = k[7]*p[2*cols+1]
        ADDS r0, r0, r1                 ;r0 = k[7]*p[2*cols+1]+k[6]*p[2*cols]+...

                LDRSB r6, [r3, r2]      ;r6 = k[8]
        ADDS r3, r3, #1                 ;r3=r3+1
        LDRB r1, [r7]           ;r1 = p[2*cols+2]
        ADDS r7, r7, #1                 ;r7=r7+1
        MULS r1, r6, r1          ;r1 = k[7]*p[2*cols+2]
        ADDS r0, r0, r1                 ;r0 = k[8]*p[2*cols+2]+k[7]*p[2*cols+1]+...


                LDR r3, [sp, #0]        ;recuperamos norm_shift
                CMP r3, #0                              ;si norm_shift <=0 salta a fin_if
                BLE     fin_if                  ;

                MOVS r1, #1             ;r1 = 1
                SUBS r2, r3, #1         ;r2 = norm_shift-1
                LSLS r1, r1, r2         ;bias = r1 = 1 << (norm_shift-1)

        CMP r0, #0                              ;si acc < 0 salta a else1
        BLT else1                               ;

if1
        ADDS r1, r0, r1                 ;r1 = acc + bias
                ASRS r1, r1, r3                         ;r1 = acc = (acc + bias) << norm_shift
        MOVS r0, r1                             ;r0 = acc
                B       fin_if

else1
        SUBS r1, r0, r1                 ;r1 = acc - bias
        ASRS r1, r1, r3                 ;r1 = acc = (acc - bias) << norm_shift
                MOVS r0, r1                             ;r0 = acc
fin_if

        CMP r0, #0                              ;si acc >= 0 salta a if2
        BGE if2                                 ;
        MOVS r0, #0                             ;devolvemos 0
        B fin_if2

if2
        CMP r0, #255                    ;si acc <= 255 salta a fin_if2
        BLE fin_if2                             ;
        MOVS r0, #255                   ;devolvemos 255
fin_if2
```

```
                LDR r6, [sp, #24]       ;recuperamos i
        MOVS r3, r6                             ;r3 = i
        MULS r3, r4, r3       ;r3 = i*cols
        LDR r2, [sp, #4]            ;recuperamos j
        ADDS r3, r3, r2             ;r3 = i*cols + j
        LDR r2, [sp, #20]           ;recuperamos out
        ADDS r2, r2, r3                 ;r2 = @out + i*cols + j
        STRB r0, [r2]          ;MemB[@out + i*cols + j] = y

        CMP r0, #0                              ;si y == 0 salta a
        BEQ no_suma                             ;
        LDR r2, [sp, #8]            ;recuperamos nonzero
        ADDS r2, r2, #1                 ;nonzero++
        STR r2, [sp, #8]            ;sp[8]=nonzero
no_suma
        LDR r2, [sp, #4]            ;recuperamos j en r2
                LDR r0, [sp, #24]       ;recuperamos i en r0
        ADDS r2, r2, #1                 ;j++
        B for_j


siguiente_i
                ADDS r0, r0, #1                         ;i++
                B for_i

fin_for
                LDR r0, [sp, #8]        ; return nonzero
                ADD sp, sp, #28

                LDR r1, =from_th
                BX r1

                ARM
from_th
                LDMIA sp!, {r4-r8, r10, r11, lr}
                BX lr

                ENDP

                END
```