
 Universidade do Minho	<b>Módulo 1</b>  <b>Introdução ao caso de estudo:</b>  <b>Convolução</b>	
--	--	---

## Motivação

Este módulo apresenta a operação de convolução entre dois sinais discretos e analisa o código *assembly* gerado por um compilador para uma versão não otimizada do operador `convolve3x1()` escrito em C. Constitui-se ainda como uma revisão da arquitectura Intel x86 e do código *assembly* gerado pelo compilador de C.

A convolução será utilizada como caso de estudo ao longo desta Unidade Curricular (UC) para ilustrar vários dos conceitos apresentados durante as aulas.

## Convolução

A área do Processamento de Imagem tem como objectivo alterar as características de uma imagem para que possa ser posteriormente tratada por processos mais sofisticados. Uma das operações fundamentais de Processamento de Imagem é a filtragem, para realizar operações como remoção de ruído, detecção dos contornos de um objecto, entre outros. Matematicamente, a **filtragem** corresponde ao operador de **convolução**.

Uma imagem é representada no mundo digital como um *array* bidimensional com uma determinada largura (W de *width*) e altura (H de *height*). Se a cor de cada ponto da imagem, designado por pixel, for representado por um inteiro, em C será algo declarado como:

```
int i[H][W];
```

O filtro  $f$  é também um *array* de valores. No caso do Processamento de Imagem estes filtros podem ser unidimensionais ou bidimensionais. No primeiro caso operam apenas sobre pixéis da mesma linha (ou coluna se o filtro for vertical), enquanto no segundo caso operam sobre uma região rectangular envolvendo pixéis de várias linhas e colunas.

Concentremo-nos agora num filtro unidimensional horizontal, com 3 colunas e 1 linha. Em C seria declarado como:

```
int f[3]; // NOTA: equivalente a int f[1][3]
```

Em termos muito simples a convolução consiste em aplicar este filtro a CADA pixel da imagem  $i$ , gerando uma nova imagem  $H$ . Matematicamente a convolução é representada pelo operador  $*$ , logo:

```
int h[H][W];
h = i * f; /* NOTA: aqui * representa a convolução
este operador não existe em C e não deve ser
confundido com a multiplicação */
```

A Figura 1 ilustra como é calculado cada elemento  $h[y][x]$ , concretizando para o caso particular de  $h[4][3]$ . Imagine que o filtro  $f[]$  é centrado no pixel  $[y][x]$  da imagem (neste caso  $i[4][3]$ ) e de seguida cada elemento de  $i[]$  que é sobreposto pelo filtro é multiplicado pelo respectivo elemento de  $f[]$ . A soma de todos estes produtos é o valor a atribuir a  $h[y][x]$ .

NOTA: na verdade existe uma subtilidade relativamente ao filtro. É suposto a soma de todos os valores de  $f[]$  ser igual a 1 – diz-se que o filtro está normalizado. Como declaramos o filtro  $f$  como sendo um *array* de inteiros dificilmente a soma dos seus elementos pode ser 1. É necessário somar os valores de todos os elementos do filtro e, antes de escrever o resultado descrito no parágrafo anterior em  $h[y][x]$ , dividi-lo por esta soma. A soma de todos os elementos do filtro chama-se a constante normalizadora.

A equação abaixo apresenta em notação matemática e para o caso de um filtro de 3x1 a expressão da convolução. Nota que esta operação é repetida para todos os pixéis da imagem.

$$h[y][x] = (f * I)[y][x] = \frac{\sum_{j=0}^2 f[j]i[y][x+j-1]}{\sum_{j=0}^2 f[j]}$$

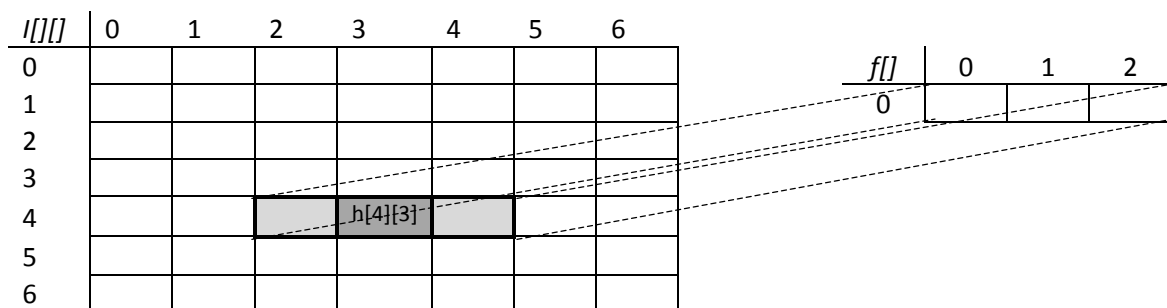


Figura 1 - região de  $i$  (imagem) usada para calcular  $h[4][3]$ : é usada uma janela com a dimensão 3x1, isto é, a mesma dimensão do filtro  $f[]$

Resumindo, o valor de  $h[y][x]$  é dado por uma média pesada da vizinhança de  $I[y][x]$ , sendo que os pesos são os valores de  $f$  e a dimensão da vizinhança é igual à dimensão de  $f$ .

```

1 static void kernel (int res[], int inp[], int ndx) {
2     res[ndx] = (inp[ndx-1] + inp[ndx] + inp[ndx+1]) / 3;
3 }
4
5 void convolve3x1 (int h[], int I[], int W, int H) {
6     register int x, y;
7
8     for (x=1 ; x<(W-1) ; x++) { // for each column of I
9         for (y=0 ; y<H ; y++) { // for each row of I
10
11             // compute h[y][x]
12             kernel (h, I, y*W+x);
13         } // y loop
14     } // x loop
15 }

```

O excerto de código acima implementa a convolução 3x1 sem qualquer optimização. Algumas notas sobre este código:

- recebe como parâmetros os apontadores para a zona de memória (*buffer*) onde deve ser guardado o resultado (*h*) e o *buffer* onde se encontra a imagem original (*l*). Recebe ainda as dimensões da imagem (*W* e *H*);
- apesar de os *buffers* *h*, e *l* conterem dados que sabemos representarem quantidades bidimensionais o acesso é feito considerando cada *buffer* como uma estrutura unidimensional. Assim *buffer[y][x]* é acedido como *buffer[y\*W+x]*, sendo *y* a linha a aceder, *x* a coluna e *W* a largura (número de colunas) da estrutura bidimensional;
- a função *kernel()* implementa o filtro. Neste caso trata-se de uma média: são somados os 3 valores de *l* na região de vizinhança de *l[y][x]* e depois divididos por 3 (constante de normalização referida acima). Nota que neste exemplo o filtro *f[]* está embebido no próprio código, em vez de representado como um *array*; não é uma solução muito flexível, mas permite-nos manter o código mais simples, para efeito de análise do *assembly*.

O resultado desta operação de filtragem depende obviamente do tipo de filtro usado. Um filtro do tipo passa-baixo elimina altas frequências, isto é, tende a suavizar as imagens diluindo os contornos dos objectos. Um exemplo de filtro passa-baixo é a média (ou *box filter*) que soma todos os elementos da vizinhança e depois divide pelo número desses elementos. Já um filtro passa-alto elimina as baixas frequências; o seu efeito é realçar os contornos dos objectos presentes numa imagem. A Tabela 1 exemplifica a aplicação de um *box filter* à imagem de Brian Kernighan (um dos inventores originais da linguagem C). A imagem original foi contaminada com ruído; os *box filters* eliminam parcialmente esse ruído. Nota que quanto maior a área do filtro, mais ruído é eliminado. Como não há almoços de borla, filtros maiores atenuam os contornos dos objectos e levam mais tempo a ser executados.

Tabela 1 - Exemplos de aplicação da convolução

	
Imagem Brian Kernighan com ruído branco	
	
Box filter (média) de 5x5	Box filter (média) de 11x11

A convolução é um operador fundamental em muitas áreas, especialmente aquelas relacionadas com o Processamento Digital de Sinal. A introdução feita neste documento simplifica muitos aspectos, cometendo mesmo algumas incorrecções para permitir esta simplificação. Para mais detalhes ver o documento disponibilizado sobre Convolução.

## Sessão Laboratorial

Nesta sessão vamos analisar o código *assembly* gerado pelo compilador de C para a arquitectura Intel x86.

Arranque a sua máquina usando a imagem do Fedora Core 13, inicie sessão usando as credenciais abaixo:

```
User: diguest
```

```
Pass: diguest
```

e inicie o gestor de janelas (`startx`).

Descarregue o código associado a este módulo e construa o executável (`make`).

O executável gerado permite aplicar a uma imagem no formato PPM (i) uma convolução com um filtro média com dimensão 3x1 ou (ii) uma convolução com um filtro Gaussiano de largura e altura U.

Corra o executável para o caso (i) escrevendo:

```
./convolve AC_images/brian_kernighan.ppm result.ppm 1
```

O último parâmetro especifica que se pretende o filtro 3x1. No ecrã foram apresentadas várias estatísticas relativas ao desempenho da máquina que serão utilizadas em sessões futuras. Visualize a imagem gerada e armazenada no ficheiro `result.ppm` (use por exemplo a aplicação `gimp` do Linux).

## Assembly

Gere o código *assembly* referente às duas rotinas apresentadas acima, escrevendo:

```
gcc -O0 -S -g convolve3x1.cpp
```

Abra o ficheiro `convolve3x1.s`, que contém o *assembly* gerado. Para facilitar a análise deste código tenha presente que o segundo argumento da directiva `.loc` indica o número da linha do código em C que corresponde às instruções *assembly* seguintes.

1. Localiza no código *assembly* as instruções que inicializam as variáveis `x` e `y` dos ciclos `for` da função `convolve3x1()`. Nota que uma vez que o código C usa o modificador `register` estas deverão estar armazenadas em registos e não em memória.
2. Localiza e explica o código *assembly* correspondente ao teste da condição do ciclo `for` mais aninhado. Nota que:
  - `H` é um parâmetro, logo deverá estar na memória, numa estrutura de dados designada por estrutura de activação. O seu endereço é dado pela soma do conteúdo do registo `%ebp` com 20;

- a instrução `cmp OP1, OP2` realiza a operação  $OP2 - OP1$  na ALU mas não guarda o resultado. Apenas os códigos de condição (*flags*) são afectados pelo resultado desta subtracção;
- a instrução `setl <reg>` coloca a 1 o registo `<reg>` se o resultado da última operação feita na ALU foi menor do que 0;
- a instrução `testb OP1, OP2` realiza a conjunção lógica (AND) entre OP1 e OP2 na ALU mas não guarda o resultado. Apenas os códigos de condição (*flags*) são afectados pelo resultado desta subtracção. De relevância neste caso é que o resultado é 0 quando  $OP1=OP2=0$ .
- a instrução `jne <label>` salta para `<label>` apenas se o resultado da última operação na ALU foi diferente de 0.

3. Localiza e explica o código assembly correspondente ao teste da condição do ciclo for menos aninhado. Nota que:

- W é um parâmetro, logo deverá estar na memória, numa estrutura de dados designada por estrutura de activação. O seu endereço é dado pela soma do conteúdo do registo `%ebp` com 16;
- a instrução `setg <reg>` coloca a 1 o registo `<reg>` se o resultado da última operação feita na ALU foi maior do que 0;

4. Desenha a estrutura de activação da função `kernel()`.

Relembra que a estrutura de activação de uma função contém o seu contexto, e é construída na pilha (*stack*) inicialmente pela função que chama esta função (neste caso será a função `convolve3x1()`) e depois pela própria função. Nota os registos e as variáveis que estamos a usar ocupam, cada uma, 4 bytes. Nota que a pilha cresce de endereços mais elevados para endereços mais baixos, isto é, a instrução `pushl OP` subtrai 4 ao `%esp`, enquanto a instrução `popl OP` adiciona 4 ao mesmo registo.

Passos da função que invoca:

- guardar registos designados de *caller saved*. No caso do x86 estes são os `%eax`, `%ecx` e `%edx`. Este passo é opcional: apenas se estes registos contiverem valores que a função invocadora necessita é que os seus valores precisam de ser guardados;
- colocar na pilha os parâmetros da função a invocar. A convenção do C é que o primeiro parâmetro a ser colocado na pilha é o parâmetro mais à esquerda da lista de parâmetros da função. A razão para isto é que o parâmetro mais à direita (primeiro na lista) fica sempre no endereço dado por 8 (`%ebp`);
- colocar na pilha o endereço de retorno, isto é o endereço da instrução que está imediatamente a seguir à instrução `call <label>`. Este passo é realizado pela própria instrução `call`.
- saltar para o endereço onde começa a função a invocar. Este passo é realizado pela própria instrução `call`.

Passos da função invocada:

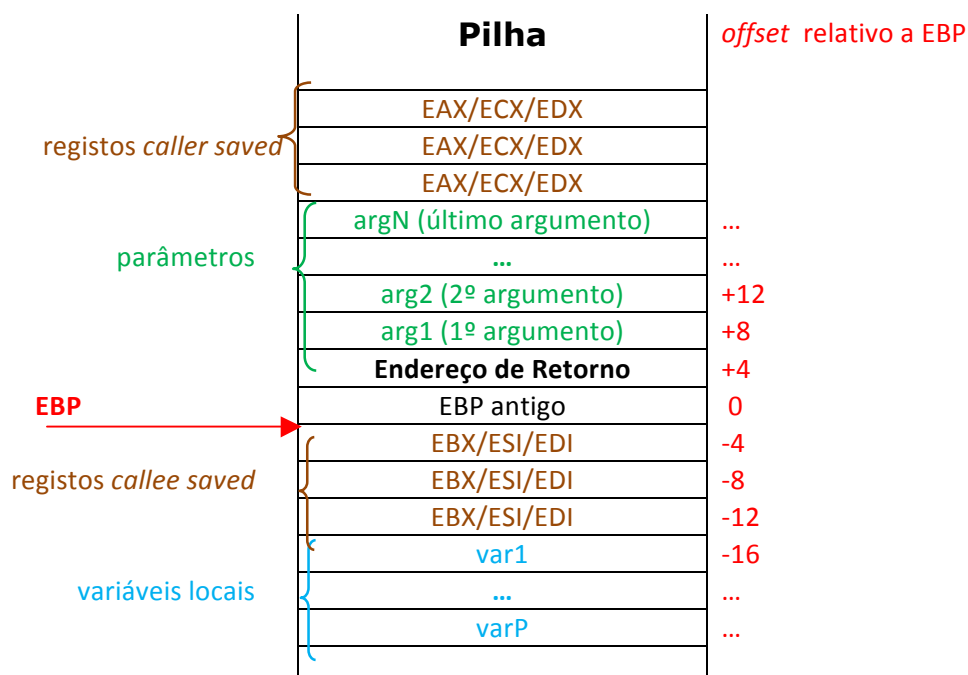
- guardar o `%ebp`. Este registo é usado como apontador fixo para a estrutura de activação. Ao guardá-lo a função invocada (`kernel()`, neste caso) está a guardar o apontador da função anterior (`convolve3x1()`, neste caso).
- actualizar o `%ebp` com o valor actual do `%esp`. Assim, o `%ebp` fica a apontar para a estrutura de activação da função `kernel()`
- guardar os registos designados de *callee saved*. No caso do x86 estes são os `%ebx`, `%esi` e `%edi`. Este passo é opcional: apenas se a função invocada usar estes registos é que deve guardar os seus valores na pilha;
- reservar espaço para as variáveis locais.

Nota que a estrutura de activação deve ser limpa quando uma função termina. Do lado da função invocada isso implica:

- libertar o espaço alocado para variáveis locais;
- repor qualquer registo *callee saved* que tenha sido guardado
- repor o `%ebp`
- ler o endereço de retorno e saltar para a instrução correspondente, o que é conseguido com a instrução `ret`

A função que invocou deve libertar o espaço ocupado pelos parâmetros e repor os valores de quaisquer registos *caller saved* que tenha guardado.

Tabela 2- estrutura de activação genérica para o x86



5. Analise a forma como são calculados os endereços e lidos de memória os valores de `inp[ndx-1]`, `inp[ndx]` e `inp[ndx+1]`. Lembre-se que:
- o cálculo do endereço de uma posição de um vector implica conhecer o endereço base (B), o índice do elemento a aceder (I) e o factor de escala (s). O endereço (E) é dado por  $E = B + I * s$
  - O IA-32 dispõe de um modo de endereçamento constituído por 4 campos: Offset (O), Base (B), Índice (I) e factor de escala (s). O e s são dados como valores imediatos (constantes) e B e I são dados como registos. Numa instrução estes aparecem no seguinte formato: O(B, I, s).  
O endereço é calculado como  $E = B + I * s + O$ .
  - A instrução LEA (*Load Effective Address*) permite calcular o endereço expresso no formato acima descrito sem aceder à memória. Esta instrução limita-se a efectuar os cálculos necessários para gerar um endereço e em alguns processadores usa uma unidade funcional específica designada por AGU (*Address Generation Unit*).