(a) e **(b)**

Resultados com FEDORA 13, 32-bits, PAPI 4.1.3.0, lab 0.12:

	Código base (a)	Com desdobramento do ciclo (b)
#I	$1{,}04 \times 10^6$	$0{,}94 \times 10^6$
TOT_ISS	$1{,}06 \times 10^6$	$1{,}06 \times 10^6$
BR_INS	$66{,}2 \times 10^3$	$33{,}8 \times 10^3$
BR_MSP	330	335
#CC	$1{,}38 \times 10^6$	$1{,}32 \times 10^6$

- O número de instruções iniciadas (***TOT_ISS***) é superior ao número de instruções efetivamente executadas (***#I***). A previsão de que os saltos são sempre executados contribui para esta diferença, porque a execução continua no endereço previsto e em determinados casos as instruções alvo do salto são inúteis tendo que ser substituídas pelas corretas.
- Nota-se que o número de saltos mal previstos é da ordem de grandeza do número de vezes que o ciclo ***for(y;;)*** é repetido → ou seja, cada execução do ciclo ***for(y;;)*** falha uma vez, na última iteração desse ciclo.
- Ao efetuar o desdobramento de ciclos o número de instruções diminui, especialmente as de salto (que são reduzidas para metade).

(c) Para que o código de ***convolve3x1()*** seja convertido automaticamente para instruções vetoriais são necessárias 3 alterações:

1. Trocar a ordem dos ciclos ***for(x;;)*** e ***for(y;;)*** : só é possível gerar instruções vetoriais quando os elementos dos vetores estão alinhados na memória (ou seja, armazenados em endereços consecutivos, por forma a serem lidos/escritos com um único "load/store").
   Com o ciclo em X mais externo e Y interno, os arrays I[] e h[] são acedidos por colunas, logo *I[y*W+x]*, *I[(y+1)*W+x]* e *I[(y+2)*W+x]* não estão guardados em endereços consecutivos.
   Com o ciclo em Y mais externo e X interno, os arrays I[] e h[] são acedidos por linhas, logo *I[y*W+x]*, *I[y*W+x+1]* e *I[y*W+x+2]* já estão guardados em endereços consecutivos.

2. Iniciar o ciclo em X em 2 (esta limitações deve-se à necessidade de alinhamento dos dados em endereços múltiplos de 4).
3. Remover a divisão por 3 na função ***kernel()*** (uma limitação do compilador).

Código original	Código alterado para vetorização

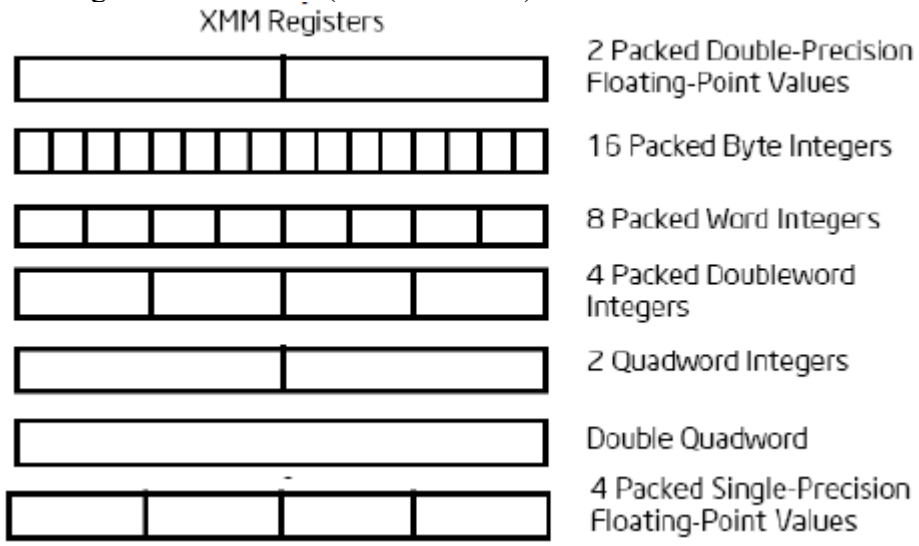
static void kernel (int res[], int inp[], int ndx) {
  res[ndx] = (inp[ndx-1] + inp[ndx] + inp[ndx+1])/3;
}
 	
void convolve3x1 (int h[], int I[], int W, int H) {
  register int x, y;
 	
  for (x=1 ; x<(W-1) ; x++) { // for each column of I[]
  for (y=0 ; y<H ; y++) {
  kernel (h, I, y*W+x);
  } // y loop
  } // x loop
}
	static void kernel (int res[], int inp[], int ndx) {
	  res[ndx] = (inp[ndx-1] + inp[ndx] + inp[ndx+1]);
	}
	
	void convolve3x1 (int h[], int I[], int W, int H) {
	  register int x, y;
	
	  for (y=0 ; y<H ; y++) { // for each row of I[]
	  for (x=2 ; x<(W-1) ; x++) {
	  kernel (h, I, y*W+x);
	  } // x loop
	  } // y loop
	}

Notas:

1. As alterações que foi necessário efetuar no código original ilustram o principal problema da "vectorização": o código tem que estar escrito de determinada forma. Para além disso não é usável em muitos algoritmos.
2. O ganho potencial no desempenho é bastante elevado (**4x** no caso MMX/SSE, mas com instruções AVX pode chegar a **8X**), no entanto, o esforço de programação também é superior a todas as técnicas estudadas anteriormente na disciplina.

SSE

- Introduz 8 registos de **128 bits** (**xmm0**..**xmm7**)



Word=16bits, Doubleword=32bits, Quadword=64bits, DoubleQuadword=128bits

- **movdqa** → mover 2 Quadword's (2x 8Bytes) entre **memória$_{128}$/ registo xmm$_i$↔ xmm$_i$**
- **movdqu** → mover 2 Quadword's (2x 8Bytes) entre **registo xmm$_i$ ↔ memória$_{128}$/xmm$_i$**

  (a - endereço alinhado num múltiplo de 16; **u** - endereço não alinhado**)**

- **paddd** → somar pares de valores de **4** bytes (**d** – **d**oubleword). O número de pares a somar é determinado pelo tamanho do registo de destino a dividir pelo tipo de dados (b|w|**d**|q=1|2|**4**|8). Se um dos operandos for memória, o endereço é alinhado. O resultado não pode ser guardado em memória.

Código não vetorial	Código vetorial	Comentário
movl -4(%esi,%ecx,4), %eax	movdq**u** (%edx), %xmm1	Lê 16 bytes ⇔ 4 inteiros
	movdq**u** (%eax,%esi), %xmm0	Lê 16 bytes ⇔ 4 inteiros
addl -8(%esi,%ecx,4), %eax	padd**d** %xmm1, %xmm0	Adiciona 4 pares de inteiros
	movdq**u** (%eax,%ebx), %xmm1	
addl (%esi,%ecx,4), %eax	padd**d** %xmm1, %xmm0	
movl %eax, (%ebx)	addl $1, %ecx	
addl $4, %ebx	movdq**a** %xmm0, 8(%edi,%eax)	Guarda resultado - 4 inteiros
	addl $16, %edx	Próximo elemento
	addl $16, %eax	Incremento de 4 inteiros
	cmpl -20(%ebp), %ecx	
	jb .L8	

Nota:
A implementação vetorial utilizada neste exemplo é semelhante a implementar um ciclo desdobrado 4 vezes.

(d) Devido à possibilidade de *memory aliasing* a vectorização do código só é correta se o vetor ***h[]*** for alocado numa zona de memória disjunta do ***I[]***. Assim, o compilador gera um teste no início da rotina para decidir, em função dos valores de ***h[]*** e ***I[]***, qual das versões a utilizar.

(e)

	Código não vetorial	Código vetorial
#I	588×10^3	198×10^3
#CC	979×10^3	432×10^3
BR_INST	65×10^3	20×10^3
VEC_INST	0	48×10^3

- Note-se que o número de saltos diminui quase 4x, uma vez que a versão vetorial processa 4 elementos por iteração.
- O tempo total (#CC) é reduzido em mais de 2x.

```
      .file "convolve3x1.cpp" # FEDORA 13 32-BITS lab 0.12
      .section    .debug_abbrev,"",@progbits
.Ldebug_abbrev0:
      .section    .debug_info,"",@progbits
.Ldebug_info0:
      .section    .debug_line,"",@progbits
.Ldebug_line0:
      .text
.Ltext0:
      .p2align 4,,15
.globl _Z11convolve3x1PiS_ii
      .type _Z11convolve3x1PiS_ii, @function
_Z11convolve3x1PiS_ii:
.LFB1:
      .file 1 "convolve3x1.cpp"
      .loc 1 35 0
      .cfi_startproc
      .cfi_personality 0x0,__gxx_personality_v0
.LVL0:
      pushl %ebp
      .cfi_def_cfa_offset 8
      movl  %esp, %ebp
      .cfi_offset 5, -8
      .cfi_def_cfa_register 5
      pushl %edi
      pushl %esi
      pushl %ebx
      subl  $44, %esp
.LBB5:
      .loc 1 38 0
      movl  20(%ebp), %edx
      testl %edx, %edx
      jle   .L15
      .cfi_offset 3, -20
      .cfi_offset 6, -16
      .cfi_offset 7, -12
      movl  16(%ebp), %eax
      .loc 1 35 0
      movl  $0, -40(%ebp)
      movl  $0, -36(%ebp)
      .loc 1 38 0
      subl  $1, %eax
      movl  %eax, -16(%ebp)
      .loc 1 35 0
      movl  16(%ebp), %eax
      sall  $2, %eax
      movl  %eax, -32(%ebp)
      movl  8(%ebp), %eax
      addl  $8, %eax
      movl  %eax, -28(%ebp)
      movl  12(%ebp), %eax
      movl  %eax, -24(%ebp)
      movl  8(%ebp), %eax
      addl  $24, %eax
      movl  %eax, -44(%ebp)
      movl  16(%ebp), %eax
      subl  $3, %eax
      movl  %eax, -48(%ebp)
      .loc 1 39 0
      shrl  $2, %eax
      movl  %eax, -20(%ebp)
```

```
      sall  $2, %eax
      movl  %eax, -52(%ebp)
      .loc 1 31 0
      addl  $2, %eax
      movl  %eax, -56(%ebp)
.LVL1:
      .p2align 4,,7
      .p2align 3
.L3:
      .loc 1 39 0
      cmpl  $2, -16(%ebp)
      jle   .L5
      .loc 1 35 0
      movl  -24(%ebp), %edx
      movl  -24(%ebp), %ebx
      movl  -24(%ebp), %esi
      addl  $4, %edx
      addl  $8, %ebx
      addl  $12, %esi
      cmpl  $5, -48(%ebp)
      jbe   .L14
      testb $15, -28(%ebp)
      jne   .L14
      movl  -24(%ebp), %eax
      addl  $20, %eax
      cmpl  %eax, -28(%ebp)
      jbe   .L20
.L12:
      movl  -24(%ebp), %eax
      addl  $24, %eax
      cmpl  %eax, -28(%ebp)
      jbe   .L21
.L13:
      movl  -24(%ebp), %eax
      addl  $28, %eax
      cmpl  %eax, -28(%ebp)
      jbe   .L22
.L10:
      .loc 1 39 0 # for (x=2 ; x<(W-1) ; x++)
      movl  -52(%ebp), %eax
      testl %eax, %eax
      je    .L23
      movl  -28(%ebp), %edi
      xorl  %eax, %eax
      xorl  %ecx, %ecx
      subl  $8, %edi
.LVL2:
      .p2align 4,,7
      .p2align 3
.L8:
      .loc 1 31 0 # res[ndx] = (inp[ndx-1] + inp[ndx] + inp[ndx+1])
      movdqu      (%edx), %xmm1
      movdqu      (%eax,%esi), %xmm0
      paddd %xmm1, %xmm0
      movdqu      (%eax,%ebx), %xmm1
      paddd %xmm1, %xmm0
      addl  $1, %ecx
      movdqa      %xmm0, 8(%edi,%eax)
      addl  $16, %edx
      addl  $16, %eax
      cmpl  -20(%ebp), %ecx
      jb    .L8
      movl  -52(%ebp), %eax
```

```
      cmpl  %eax, -48(%ebp)
      movl  -56(%ebp), %esi
      je    .L5
.LVL3:
.L7:
      movl  -40(%ebp), %ecx
      xorl  %ebx, %ebx
      movl  8(%ebp), %eax
      leal  (%esi,%ecx), %ecx
      leal  (%eax,%ecx,4), %edi
      .p2align 4,,7
      .p2align 3
.L9:
.LVL4:
      .loc 1 35 0
      movl  12(%ebp), %edx
      .loc 1 39 0
      addl  $1, %esi
.LVL5:
      .loc 1 35 0
      addl  %ebx, %edx
      .loc 1 39 0
      addl  $4, %ebx
.LBB6:
.LBB7:
      .loc 1 31 0
      movl  (%edx,%ecx,4), %eax
      addl  -4(%edx,%ecx,4), %eax
      addl  4(%edx,%ecx,4), %eax
      movl  %eax, (%edi)
.LBE7:
.LBE6:
      .loc 1 39 0
      addl  $4, %edi
      cmpl  %esi, -16(%ebp)
      jg    .L9
.LVL6:
.L5:
      .loc 1 38 0
      movl  -32(%ebp), %eax
      addl  %eax, -28(%ebp)
      movl  16(%ebp), %eax
      addl  %eax, -40(%ebp)
      movl  -32(%ebp), %eax
      addl  %eax, -24(%ebp)
      addl  %eax, -44(%ebp)
      addl  $1, -36(%ebp)
.LVL7:
      movl  20(%ebp), %eax
      cmpl  %eax, -36(%ebp)
      jne   .L3
.LVL8:
.L15:
.LBE5:
      .loc 1 44 0
      addl  $44, %esp
      popl  %ebx
      .cfi_remember_state
      .cfi_restore 3
      popl  %esi
      .cfi_restore 6
      popl  %edi
      .cfi_restore 7
```

```
      popl  %ebp
      .cfi_restore 5
      .cfi_def_cfa 4, 4
      ret
.LVL9:
      .p2align 4,,7
      .p2align 3
.L21:
      .cfi_restore_state
.LBB10:
      .loc 1 35 0
      cmpl  %ebx, -44(%ebp)
      jb    .L13
      .p2align 4,,7
      .p2align 3
.L14:
      movl  -28(%ebp), %ebx
      movl  $3, %edx
      movl  -40(%ebp), %edi
      movl  12(%ebp), %esi
.LVL10:
      .p2align 4,,7
      .p2align 3
.L4:
      leal  (%edx,%edi), %ecx
.LBB9:
.LBB8:
      .loc 1 31 0 # res[ndx] = (inp[ndx-1] + inp[ndx] + inp[ndx+1])
      addl  $1, %edx
      movl  -4(%esi,%ecx,4), %eax
      addl  -8(%esi,%ecx,4), %eax
      addl  (%esi,%ecx,4), %eax
      movl  %eax, (%ebx)
      addl  $4, %ebx
.LBE8:
.LBE9:
      .loc 1 39 0
      cmpl  16(%ebp), %edx
      jne   .L4
      jmp   .L5
.LVL11:
      .p2align 4,,7
      .p2align 3
.L20:
      .loc 1 35 0
      cmpl  %edx, -44(%ebp)
      jb    .L12
      .p2align 4,,8
      jmp   .L14
      .p2align 4,,7
      .p2align 3
.L22:
      cmpl  %esi, -44(%ebp)
      .p2align 4,,5
      jb    .L10
      .p2align 4,,8
      jmp   .L14
.L23:
      .loc 1 39 0
      movl  $2, %esi
      .p2align 4,,3
      jmp   .L7
[ ... ]
```