Using MMX™ Instructions to Compute the L1 Norm Between Two 16-bit Vectors

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Note: Please be aware that the software programming article below is posted as a public service and may no longer be supported by Intel.

Copyright © Intel Corporation 2004
* Other names and brands may be claimed as the property of others.
CONTENTS

1.0. INTRODUCTION
2.0. THE L1 NORM FUNCTION
   2.1. Core of the MMX Technology L1 Norm
   2.2. Scheduling for Optimum Pairing
3.0. MMX CODE PERFORMANCE
4.0. CODE LISTING: L1 Norm
1.0. INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents an example of code that exploits these instructions. Specifically, an MMX™ technology implementation of a 16-bit L1-norm is described.

2.0. THE L1 NORM FUNCTION

The L1 Norm function, also known as "sum of absolute differences", takes two vectors \( x \) and \( y \) and computes

\[
L1Norm = \sum |x_i - y_i|.
\]

In this application note, the input vectors \( x \) and \( y \) are arrays of 16-bit words, but the sum is 32 bits, in order to provide enough dynamic range to represent the sum accurately without saturation or overflow. If the input values are small, it may be possible to accumulate in 16 bits, in which case a more efficient routine that the one described in this note could be constructed.

2.1. Core of the MMX Technology L1 Norm

The method used to code an L1 Norm in MMX instructions consists of the following three steps:

1. Compute \( |x_i - y_i| \) using 16-bit SIMD, producing four 16-bit results. Note that these values are all positive (unsigned) numbers.

2. Use the Unpack instructions to expand the 16-bit results to 32 bits.

3. Add the four 32-bit values to an accumulator register, using two SIMD adds (each of which operates on a pair of 32-bit values).

This procedure is depicted in Figure 1.
Using MMX™ Instructions to Compute the L1 Norm Between Two 16-bit Vectors

March 1996

Figure 1. Core of L1 Norm Algorithm

Since there is no MMX technology absolute-value instruction, the step of computing \(|x - y|\) from \((x - y)\) is a little tricky. The way to compute an absolute value is to exploit the fact that in two's-complement arithmetic, the negative of a number \(z\) is equal to the 1's complement of \(z\) plus 1. Therefore, if MMX register MM4 contains four arbitrary signed 16-bit words, the absolute value of all four words can be taken using the following sequence of code:

```
MOVQ mm5,MM4 ;make copy of MM4
PSRAW MM4,15 ;replicate sign bit
PXOR mm5,MM4 ;do 1's complement
PSUBSW mm5,MM4 ;add 1
```

The PSRAW instruction in this sequence produces 0xffff in those fields of MM4 which are negative, and 0 in those fields of MM4 which are positive. The PXOR exclusive-ORs this with MM4, which leaves the positive values unchanged but one's-complements the negative values. The PSUBS again leaves the positive values unchanged but adds 1 (by subtracting -1) from values in the negative fields. Therefore, just the negative fields have been negated, leaving the absolute value in all fields.

**Example 1. Core of the L1 Norm Algorithm**

```
movq mm1,[esi+8*ecx] ;1 Load next chunk of x
psubsw mm1,[edi+8*ecx] ;2 Subtract y from mem,w/signed sat
movq mm2,mm1
psraw mm1,15
pxor mm2,mm1
psubsw mm2,mm1
movq mm1,mm2
punpcklwd mm2,mm7
paddd mm0,mm2
punpckhwd mm1,mm7
paddd mm0,mm1
```

; 1 Load next chunk of x
; 2 Subtract y from mem,w/signed sat
; 3 Make absolute value (see text)
; 4
; 5
; 6
; 7 Make a copy of a b c d
; 8 mm7=0. mm2 now has 0 c 0 d
; 9 Add to accumulator (mm0)
; 10 mm1 now has 0 a 0 b
; 11 Add to accumulator
The complete series of instructions for one iteration of the L1 Norm is shown in Example 1. Computation of four terms of the L1 Norm requires 11 MMX instructions.

Observe that the two PUNPCK instructions require a register (MM7) to be preloaded with the value zero. This is due to the way in which the PUNPCK instruction works—it interleaves the fields of the two input register. In order to do a simple zero-extended unpack, one of the registers needs to be zero.

2.2. Scheduling for Optimum Pairing

In order to get the best performance, the code needs to be properly interleaved in order to avoid read-after-write dependencies and to satisfy the pairing rules of the Pentium® processor, which allow only one shift/pack/unpack and only one memory access per clock. (There are other rules which do not affect this particular algorithm.)

Referring to the numbering of the instructions shown in the comments of Example 1, it is possible to interleave two iterations of the algorithm so that perfect U- and V-pipe pairing is achieved, as shown in Example 2.

Example 2. L1 Norm with U,V Pairing

```
1 ; prelude to the loop
2
3
loop:
4 1 ; paired in U,V pipe
5 2 ; ditto
6 3 ; etc.
7 4
8 5
9 6
10 7
11 8
1 9
2 10
3 sub ecx,2
11 jge loop
```

This sequence of code takes advantage of the fact that an MMX instruction and a pairable integer instruction can be paired. This permits the loop control to be split across the last two instructions, which in turn means that the decrement of ECX takes place two instructions before the top of the loop, which avoids an AGI penalty.
3.0. MMX CODE PERFORMANCE

This section describes the performance of the MMX technology L1 Norm and compares it to a traditional implementation using Intel Architecture integer instructions.

The loop shown in Section 2.2 computes eight terms of the L1 Norm, since each basic sequence (shown in Example 2) computes four terms and two sequences have been interleaved. The loop takes 12 clocks, therefore the speed of the MMX technology L1 norm is:

\[ \text{MMX L1 Norm} = 1.5 \text{ clocks per term} \]

Computing an L1 norm with regular integer instructions is non-trivial, for the same reason as the MMX technology implementation: there is no absolute value instruction. In order to avoid the penalties of mispredicted-predicted branches, a similar technique to the one use for MMX technology can be employed (the one's-complement-plus-one trick). This technique requires about 10 instructions to compute one term of the L1 norm. This code can be paired so that two instructions execute per clock. Thus, the speed of the integer L1 norm is:

\[ \text{Scalar L1 Norm} = 5 \text{ clocks per term} \]

The MMX technology implementation of 16-bit L1 norm is therefore about 3.3 times faster than scalar integer code.
4.0. CODE LISTING: L1 Norm

```
.L486p
.model flat, c
.code
; L1 norm on 16-bit vectors using MMX code. For simplicity, this
; routine REQUIRES the arrays to be a multiple of 8 (words) in size.
; This is because one iteration handles 4 words, and we have interleaved
; two iterations for maximum utilization of the U,V pipes.
; Handling arbitrary length is left as an exercise for the reader...
l1norm PROC C PUBLIC USES esi edi ebx
    in1:DWORD, in2:DWORD, len:DWORD
    ; in1, in2 are word arrays
    ; Len is number of words
    mov esi, in1
    mov ecx, len
    mov edi, in2
    shr ecx, 2
    dec ecx
    pxor mm0, mm0             ; set accumulator to zero
    pxor mm7, mm7             ; mm7=0, for use in unpacking
    ; The basic algorithm is a sequence of 11 MMX instructions. Here
    ; we "get the pipeline going" by doing the first 3 instructions of
    ; the sequence. Then we fall into a loop in which two copies of
    ; the sequence are interleaved.
    movq mm3, [esi+8*ecx]     ; load from in1
    psubsw mm3, [edi+8*ecx]   ; subtract in2
    movq MM4, mm3             ; copy a-b
    dec ecx
    loop1:
    movq mm1, [esi+8*ecx]     ; load from in1
    psraw mm1, 15             ; replicate sign bits
    psubw mm1, [edi+8*ecx]    ; subtract in2
    pxor MM4, mm3             ; 1's complement of negative fields
    psubsw MM4, mm3           ; now MM4 has absolute values
    movq mm2, mm1             ; copy a-b
    movq mm3, MM4             ; copy a b c d
    psraw mm1, 15             ; replicate sign bits
    punpcklwd MM4, mm7        ; now MM4 has 0 c 0 d
    pxor mm2, mm1             ; 1's complement of negative fields
    paddd mm0, MM4             ; add to accum
    psubsw mm2, mm1           ; now mm2 has absolute values
    punpckhwd mm3, mm7        ; now mm3 has 0 a 0 b
    movq mm1, mm2             ; copy a b c d
    paddd mm0, mm3             ; add to accum
    punpckhwd mm2, mm7        ; now mm2 has 0 c 0 d
    movq mm3, [esi+8*ecx-8]   ; load from in1: START OF 2nd SEQUENCE
    paddd mm0, mm2             ; add to accum
    psubbw mm3, [edi+8*ecx-8] ; subtract in2
    punpckhwd mm1, mm7        ; now mm1 has 0 a 0 b
    movq MM4, mm3             ; copy a-b
    sub ecx, 2                ; add the two dwords in the accumulator together
    paddd mm0, mm1             ; add to accum
    jge loop1
    movq mm1, mm0
    psrlq mm1, 32
    paddd mm0, mm1
    movdf eax, mm0
    emms
    ret
l1norm EndP
END
```