Block-Matching In Motion Estimation Algorithms Using Streaming SIMD Extensions 3

Version 2

12/2003
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.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Pentium® II processors, Pentium® III processors and Pentium® 4 processors may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

† Third-party brands and names are the property of their respective owners.

Copyright © Intel Corporation 2002, 2003
# Table of Contents

1. Introduction ................................................................................................................................. 5
2. Block-Matching ................................................................................................................................... 5
   2.1 Applications for Block-Matching ................................................................................................ 5
   2.2 Implementing the Block-Matching Algorithm ............................................................................ 6
      2.2.1 Techniques .......................................................................................................................... 7
      2.2.2 Using the new LDDQU Instruction ..................................................................................... 8
3. Performance ........................................................................................................................................... 8
   3.1 Gains/Improvements ................................................................................................................. 8
   3.2 Considerations ............................................................................................................................ 9
4. Conclusion ........................................................................................................................................... 9
5. C/C++ Coding Example .................................................................................................................... 10
6. SSE3 Intrinsics Code Example .......................................................................................................... 12
Appendix A - Performance Data ........................................................................................................... A-1
   Performance Data Revision History ................................................................................................ A-1
   Test Systems Configuration ............................................................................................................. A-1
**Revision History**

<table>
<thead>
<tr>
<th>Revision</th>
<th>Revision History</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Original publication of document</td>
<td>7/2003</td>
</tr>
<tr>
<td>2.0</td>
<td>Update performance data</td>
<td>12/2003</td>
</tr>
</tbody>
</table>

**References**

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.


1 Introduction

The Streaming SIMD Extensions 3 (SSE3) for IA-32 Intel® Architecture accelerates performance of Streaming SIMD Extensions 2 (SSE2) technology, Streaming SIMD Extensions (SSE) technology and scientific applications over the current Intel Pentium® 4 processor generation. SSE3 introduces new instructions that significantly improve complex arithmetic evaluation, evaluations of quantities like dot products, conversion to integer for applications using x87-FP code, and 128-bit unaligned memory accesses. In addition, SSE3 has added support for better synchronization between multiple agents over the current Pentium® 4 generation.

The SSE2 technology introduced new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced with SSE. The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application. This application note demonstrates specifically an application of the 128-bit integer SIMD instructions, and includes examples of code that exploit the SSE2 instructions.

As an example of the usage of the 128-bit version of the psadbw and paddw instructions, this application note gives details on coding a fast block-matching algorithm. This type of code is used in MPEG and MPEG2 encoders and accounts for approximately 40-70% of execution time of the encoders.

A previous application note, AP-940, “Block-Matching In Motion Estimation Algorithms Using Streaming SIMD Extensions 2 (SSE2)” is very similar to this application notes with one exception. This application notes uses SSE3 for 128-bit unaligned loads to further increase the performance of video encoding.

2 Block-Matching

The block-matching algorithm compares two 16-byte by 16-byte blocks of memory by accumulating either a sum of the absolute differences (SAD) between corresponding pixels or a squared sum of differences (SSD). This paper discusses only the more commonly used SAD algorithm. The final summation is an accurate measure of how well two blocks of video match. Note that this algorithm requires a branch for the absolute difference calculation for each pair of pixels, 256 overall, of the form:

```
if(difference < 0) difference *= -1;
running_total += difference;
```

2.1 Applications for Block-Matching

In MPEG encoders, there are two methods of achieving compression: within a frame and over a sequence of frames. Similar to JPEG compression, each individual frame of a video sequence can be compressed individually. This compressed data, usually 3-4 times smaller than the original, is then stored.

The compression ratios can be dramatically increased by noting that consecutive frames of video are usually nearly identical. The differences are often made up of blocks of pixels moving around the frame in an orderly manner. Think of a video of a tennis match: the majority of the pixels slide back and forth
across the frame, but they do not change. The ball still looks like a ball; it is just in different positions from frame to frame.

Instead of storing compressed versions of the blocks, it is possible to store the motion vector instead. For example, if a particular 16 by 16 block has moved 2 pixels left, and 1 pixel down, it is much more efficient to store this information than to store compressed data for those 256 pixels.

2.2 Implementing the Block-Matching Algorithm

Typically, the block-matching algorithm is embedded into a motion estimation (ME) search algorithm. This search can be done in many different ways, but the simplest is a full search. This is an exhaustive comparison of all blocks in a specified range, pixel by pixel. For example, a common selection would be a 16 by 16 full search, meaning that the current block is compared to 256 reference blocks. The reference block is compared at a starting point, pixel by pixel, with the current block. Then the reference block is moved over by a pixel and compared against the same current block. This continues through the search area. Wherever the lowest SAD is found is considered the best match.

Figure 1: Comparing the First Set of Blocks

Figure 1 shows the first pair of blocks to be compared. The motion estimation search begins at the upper left-hand corner of the search range. Each corresponding pixel in the two blocks is subtracted, and the absolute difference accumulated. This sum of absolute differences gives a numeric value for the relative degree of difference between the two blocks. The smaller the SAD value, the better the block match.
2.2.1 Techniques

The first step in the speedup of the ME algorithm is to reduce the number of block comparisons required. Techniques such as sub-sampling, logarithmic searches, and telescopic searches can be used to improve the efficiency of the encoder. Reductions on the order of 3-8X can be made while maintaining acceptable video quality. See the references section of this application note for books that provide examples of these techniques.

The second step in speedup of the ME algorithm came with the introduction of the Pentium® III processor with SSE and specifically with the _psadbw_ instruction. This instruction compares eight bytes from each of the two blocks at once, returning a single SAD value. Not only does a single instruction replace the eight subtractions and accumulations, but it also handles the absolute value determination without the use of branches. This produces a speedup of about 1.7X over an MMX™ technology implementation on the Pentium III processor.

With the addition of the SSE2 instructions, this instruction has been widened to operate on 16 pixels at once. This allows you to perform a SAD operation on an entire row of the block with one instruction.

Some algorithms use an “early return” threshold. If, after a certain number of rows have been compared, the current accumulated SAD value is over a threshold value, the algorithm is aborted. This is good practice since we normally don’t want to keep comparing blocks if they are very different than previous

Figure 2 shows the second two blocks to be compared. Note that there is much overlap between this block-matching pair and the first pair, even though different pixels are compared in each case. This comparing process continues through the entire search range. Obviously, this search can quickly add up to millions of cycles as the two arrays of 256 values are compared 256 times! And of course, this cycle must be repeated for each block in the frame (typically 720 X 480 pixels for MPEG2, yielding 45 * 30 = 1350 blocks per frame). This is why motion estimation utilizes the majority of the execution time in an encoder.
blocks. Using the `psadbw` instruction of either 8-byte or 16-byte width makes this technique less useful in most algorithms. It usually takes longer to decide to exit (which requires a branch mispredict penalty) than to complete the block-matching function; it decreases the accuracy as well. As processing speeds increase, branch misprediction recovery times have typically become relatively longer. As a result, the value of 'early return' has been steadily declining and on future processors may prove to be a significant detriment to motion estimation performance.

2.2.2 Using the new LDDQU Instruction

Because the search moves across the frame on a pixel by pixel basis, it is natural to incur cache line splits when executing a block-matching algorithm. A cache line split occurs when a data item resides on two cache lines, i.e., is split across two cache lines. When accessed, the processor must read both cache lines and in addition, incurs a penalty to handle the special access.

With the addition of SSE3, replacing the `movdqu` instruction with the `lddqu` instruction avoids the cache line split. The `lddqu` instruction loads 128 bits. Consider the following instruction:

```
lddqu xmm, m128
```

If `m128` is aligned to a 16-byte boundary, the 16 bytes of the data at memory location `m128` are loaded to the 128-bit register `xmm`. If `m128` is not aligned to a 16-byte boundary, the processor loads a 32-byte block starting at the 16-byte aligned address immediately below the address of the load request, and extracts from the 32-byte block the requested 16 bytes. Using `lddqu` alone provides a significant performance increase in the block matching implementation presented in this paper.

3 Performance

Using the SSE3 128-bit unaligned load instruction, LDDQU to implement block-matching speeds up the performance relative to earlier technologies. This section describes the gains and improvements achieved using the SSE3 instruction and addresses related programming considerations.

3.1 Gains/Improvements

The block-matching algorithm was previously optimized for SSE2 technology in AP-940. The SSE3 128-bit unaligned load instruction LDDQU is used to further optimize the implementation in this application note.

The speedup factors are:

- 128-bit unaligned instruction designed to avoid cache line splits
- Intel’s Pentium® 4 processor with SSE3 micro-architecture improvements including:
  — Larger caches (but access time is slightly slower)

The benefits of the new SSE3 128-bit unaligned load instruction, LDDQU are greater than those resulting from the new architecture for this block-matching algorithm.

Since this algorithm uses large amounts of memory and constantly thrashes the caches (each of the two frames is 720 X 480, or more than 300K), it requires large amounts of bus bandwidth and CPU memory subsystem resources. Using prefetch instructions gave us no gain, since the prefetches tended to displace higher priority loads. Because the amount of time spent in calculations was low compared to the number of loads, prefetches are not suitable in this case. During the search, as we move pixel by pixel across the search range, we would normally see data cache splits on the Pentium 4 processor when using the
movdqu instruction. When using the new SSE3 128-bit unaligned load instruction, LDDQU we avoid the data cache splits that we incurred with the Pentium 4 processor.

### 3.2 Considerations

If you are unsure if your implementation would benefit from using the new instruction, you can use the Intel® VTune™ Performance Analyzer to help you analyze your implementation. Using the sampling feature, collect the `clockticks` event in addition to `Split Loads Retired`. Then use the VTune™ Performance Analyzer’s Tuning Assistant feature which can tell you the estimated impact (in seconds) that split loads is having on your application. You can then determine if using the new `lddqu` instruction is worth the recoding effort.

Also, keep in mind that at this time, the Intel C/C++ compiler does not use the `lddqu` instruction automatically but Intel will be investigating the possibility of using it automatically in the future.

### 4 Conclusion

The new SSE3 128-bit unaligned load instruction, LDDQU can provide a significant performance gain in the block-matching motion estimation algorithm for MPEG and MPEG2 encoders.

The following architecture improvements contribute to the overall performance gain:

- the `lddqu` instruction

In addition, the following micro-architecture improvements contribute to performance gain:

- Larger caches
5 C/C++ Coding Example

// SAD-based block matching procedure for single macroblock
// Search area is specified by iLeft, iTop, iRight, iBottom.
// Result is returned through piMvPos.
// ARG:  pucRef  Reference frame left top pointer.
//       pucCur  Current frame left top pointer.
//       piMvPos Pointer of position indicated by result MV.
//       iWidth  Width of the frame.
//       iLeft  Search area left position.
//       iRight Search area right position.
//       iTop  Search area top position.
//       iBottom  Search area bottom position.
//       iHpos  Current block left position in the frame.
//       iVpos  Current block top position in the frame.
// Comment:
//   MV  1 means 1pel
//   Not check validity of search area.

iDown = iWidth - 16;
/* Set start point for Reference and Search window */
pucCurLeftTop = pucCur + (iVpos * iWidth) + iHpos;
/* Matching loop */
for(iY = iTop; iY <= iTBottom; iY++) {
  /* Set start point for Reference and Search window */
pucRefLeft = pucRef + (iY * iWidth);
  for(iX = iLeft; iX <= iRight; iX++) {
    /* Init temporal AD */
    iTmpAd = 0;
    /* Copy Left position pointer */
pucC = pucCurLeftTop;
pucR = pucRefLeft + iX;
    /* Get absolute difference of current position */
    for(iV = 0; iV < 16; iV++) {
      for(iH = 0; iH < 16; iH++) {
        iTmpAd += abs((INT32)*(pucC++) - (INT32)*(pucR++));
      }
    }
    /* Current pointer down 1 line */
    pucC += iDown;
pucR += iDown;
  }
  /* Check minimum AD */
  if(iMinAd > iTmpAd) {
    iMinAd = iTmpAd;
    *piMvPos = iX;
    *(piMvPos+1) = iY;
  }
} } return iMinAd;
6 SSE3 Intrinsics Code Example

```c
pucRef = (pucRef + (iTop * iWidth) + iLeft);
pucC = pucCur + (iVpos * iWidth) + iHpos;

// Block loop
for (iY = 0; iY <= (iBottom - iTop); iY++) {

    // Set start point for Reference window
    pucR = pucRef + iY * iWidth;

    for (iX = 0; iX <= (iRight - iLeft); iX++) {
        sum = _mm_xor_si128(sum, sum);   // Clear accumulator
        sum2 = _mm_xor_si128(sum2, sum2);  // Clear accumulator

        // Get SAD for block pair
        row2 = _mm_lldqiu_si128((__m128i *)pucR);
        row4 = _mm_lldqiu_si128((__m128i *) (pucR + iWidth));
        row6 = _mm_lldqiu_si128((__m128i *) (pucR + 2*iWidth));
        row8 = _mm_lldqiu_si128((__m128i *) (pucR + 3*iWidth));
        row1 = _mm_load_si128((__m128i *) pucC);
        row3 = _mm_load_si128((__m128i *) (pucC + iWidth));
        row5 = _mm_load_si128((__m128i *) (pucC + 2*iWidth));
        row7 = _mm_load_si128((__m128i *) (pucC + 3*iWidth));

        row1 = _mm_sad_epu8(row1, row2);
        row3 = _mm_sad_epu8(row3, row4);
        sum = _mm_add_epi16(sum, row1);
        sum2 = _mm_add_epi16(sum2, row3);

        row5 = _mm_sad_epu8(row5, row6);
        row7 = _mm_sad_epu8(row7, row8);
        sum = _mm_add_epi16(sum, row5);
        sum2 = _mm_add_epi16(sum2, row7);

        row2 = _mm_lldqiu_si128((__m128i *) (pucR + 4*iWidth));
        row4 = _mm_lldqiu_si128((__m128i *) (pucR + 5*iWidth));
        row6 = _mm_lldqiu_si128((__m128i *) (pucR + 6*iWidth));
        row8 = _mm_lldqiu_si128((__m128i *) (pucR + 7*iWidth));
        row1 = _mm_load_si128((__m128i *) (pucC + 4*iWidth));
        row3 = _mm_load_si128((__m128i *) (pucC + 5*iWidth));
        row5 = _mm_load_si128((__m128i *) (pucC + 6*iWidth));
        row7 = _mm_load_si128((__m128i *) (pucC + 7*iWidth));
    }
}
```
row1 = _mm_sad_epu8(row1, row2);
row3 = _mm_sad_epu8(row3, row4);
sum = _mm_add_epi16(sum, row1);
sum2 = _mm_add_epi16(sum2, row3);
row5 = _mm_sad_epu8(row5, row6);
row7 = _mm_sad_epu8(row7, row8);
sum = _mm_add_epi16(sum, row5);
sum2 = _mm_add_epi16(sum2, row7);
row2 = _mm_lddqu_si128((__m128i *)(pucR + 8*iWidth));
row4 = _mm_lddqu_si128((__m128i *)(pucR + 9*iWidth));
row6 = _mm_lddqu_si128((__m128i *)(pucR + 10*iWidth));
row8 = _mm_lddqu_si128((__m128i *)(pucR + 11*iWidth));
row1 = _mm_load_si128((__m128i *) (pucC + 8*iWidth));
row3 = _mm_load_si128((__m128i *) (pucC + 9*iWidth));
row5 = _mm_load_si128((__m128i *) (pucC + 10*iWidth));
row7 = _mm_load_si128((__m128i *) (pucC + 11*iWidth));
row1 = _mm_sad_epu8(row1, row2);
row3 = _mm_sad_epu8(row3, row4);
sum = _mm_add_epi16(sum, row1);
sum2 = _mm_add_epi16(sum2, row3);
row5 = _mm_sad_epu8(row5, row6);
row7 = _mm_sad_epu8(row7, row8);
sum = _mm_add_epi16(sum, row5);
sum2 = _mm_add_epi16(sum2, row7);
row2 = _mm_lddqu_si128((__m128i *)(pucR + 12*iWidth));
row4 = _mm_lddqu_si128((__m128i *)(pucR + 13*iWidth));
row6 = _mm_lddqu_si128((__m128i *)(pucR + 14*iWidth));
row8 = _mm_lddqu_si128((__m128i *)(pucR + 15*iWidth));
row1 = _mm_load_si128((__m128i *) (pucC + 12*iWidth));
row3 = _mm_load_si128((__m128i *) (pucC + 13*iWidth));
row5 = _mm_load_si128((__m128i *) (pucC + 14*iWidth));
row7 = _mm_load_si128((__m128i *) (pucC + 15*iWidth));
row1 = _mm_sad_epu8(row1, row2);
row3 = _mm_sad_epu8(row3, row4);
sum = _mm_add_epi16(sum, row1);
sum2 = _mm_add_epi16(sum2, row3);
row5 = _mm_sad_epu8(row5, row6);
row7 = _mm_sad_epu8(row7, row8);
```c
sum = _mm_add_epi16(sum, row5);
sum2 = _mm_add_epi16(sum2, row7);
sum = _mm_add_epi16(sum, sum2);

// Check for new minimum SAD
iTmpAd = _mm_cvtsi128_si32(sum);
if(iTmpAd < iMinAd){
    iMinAd = iTmpAd;
    *(piMvPos) = iX + iLeft;
    *(piMvPos+1) = iY + iTop;
}
pucR++;
}
return iMinAd;
```
Appendix A - Performance Data

Performance Data Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Revision History</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Original publication of document</td>
<td>7/2003</td>
</tr>
<tr>
<td>2.0</td>
<td>Update performance data</td>
<td>12/2003</td>
</tr>
</tbody>
</table>

Table 1: Performance Data of ME Implementations

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4 Processor (2.8 GHz)</th>
<th>Pentium 4 Processor with SSE3 (2.8 GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE2 instructions</td>
<td>5296</td>
<td>5156</td>
</tr>
<tr>
<td>SSE3 instructions (Iddqu)</td>
<td>-</td>
<td>3750</td>
</tr>
</tbody>
</table>

Table 2: Speedups from Table 1 Performance Data

<table>
<thead>
<tr>
<th>Implementations and Platforms</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4 processor (SSE3 vs. SSE2)</td>
<td>1.37</td>
</tr>
</tbody>
</table>

Performance was measured using an Intel Pentium 4 2.8 GHz processor with SSE2 and an Intel Pentium 4 2.8GHz processor with SSE3. See Test Systems Configuration below for a detailed description of the test systems.

Test Systems Configuration

Table 3: Pentium 4 SSE2 Configuration

<table>
<thead>
<tr>
<th>Processor</th>
<th>Pentium 4 Processor at 2.8 GHz with SSE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Intel® Desktop Board Springdale (Chelan – C23691-200)</td>
</tr>
<tr>
<td>Bios Version</td>
<td>VVPSGI5A.86A.0033.D.0304101021</td>
</tr>
<tr>
<td>Secondary Cache</td>
<td>512KB</td>
</tr>
<tr>
<td>Memory Size</td>
<td>512MB DDR 333MHz</td>
</tr>
<tr>
<td>Ultra ATA Storage Driver</td>
<td>Intel 5.0.10020</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>Maxtor 6L080J4</td>
</tr>
<tr>
<td>Video Controller/Bus</td>
<td>ATI Radeon 9700 Pro – 128MB</td>
</tr>
</tbody>
</table>
Table 4: Pentium 4 SSE3 Configuration

<table>
<thead>
<tr>
<th>Processor</th>
<th>Pentium 4 Processor at 2.8 GHz with SSE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Intel® Desktop Board Springdale (Chelan – C23691-200)</td>
</tr>
<tr>
<td>Bios Version</td>
<td>VVPGL5A.86A.0033.D.0304101021</td>
</tr>
<tr>
<td>Secondary Cache</td>
<td>512KB</td>
</tr>
<tr>
<td>Memory Size</td>
<td>512MB DDR 333MHz</td>
</tr>
<tr>
<td>Ultra ATA Storage Driver</td>
<td>Intel 5.0.10020</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>Maxtor 6L080J4</td>
</tr>
<tr>
<td>Video Controller/Bus</td>
<td>ATI Radeon 9700 Pro – 128MB</td>
</tr>
<tr>
<td>Video Driver Revision</td>
<td>5.1.2600.0</td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows XP Professional 2002 SP1</td>
</tr>
</tbody>
</table>