International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:4, 2008

Performance Enhancement of Motion Estimation Using SSE2 Technology

Trung Hieu Tran, Hyo-Moon Cho, and Sang-Bock Cho

***Abstract*—**Motion estimation is the most computationally intensive part in video processing. Many fast motion estimation algorithms have been proposed to decrease the computational complexity by reducing the number of candidate motion vectors. However, these studies are for fast search algorithms themselves while almost image and video compressions are operated with software based. Therefore, the timing constraints for running these motion estimation algorithms not only challenge for the video codec but also overwhelm for some of processors. In this paper, the performance of motion estimation is enhanced by using Intel's Streaming SIMD Extension 2 (SSE2) technology with Intel Pentium 4 processor.

***Keywords*—**Motion Estimation, Full Search, Three Step Search, MMX/SSE/SSE2 Technologies, SIMD.

I. Introduction

VIDEO coding standards like MPEG-1/2/4/7 by the Moving Picture Experts Group (MPEG) and H.26x-1/2/3/4 by the Video Coding Experts Group (VCEG) have been used in many applications such as video conferencing, video on demand, video phones, and so on [1].

In image and video compression, motion estimation and motion compensation are key functions. A 1/4-pixel motion estimation is used to increase the accuracy in H.264 video standard, whereas a 1/2-pixel motion estimation is used in MPEG-4 video standard. Using 1/4-pixel motion estimation, the computational complexity increases four times but the high compression efficiency will be achieved. To solve this computational complexity problem, the fast search algorithms based on TSS (Three Step Search) have been proposed and widely used [2]. The timing constraints with such high data rates not only challenge enough even for the video codec but also overwhelm for some of superscalar processors [3]. Performing these operations with real time is not easy on most platforms if image resolutions at acceptable quality are desired.

Motion estimation algorithm is the most important factor in image and video compression processing. And it consists of repetitive operations which could benefit greatly from some architecture to perform repetitive tasks efficiently.

Trung Hieu Tran, Master student, is with the School of EE, University of Ulsan, Ulsan, 680-749 South Korea (corresponding author phone: 82-52-259-1629; e-mail: tthieu@mail.ulsan.ac.kr).

Hyo-Moon Cho, PhD Candidate, is with the School of EE, University of Ulsan, Ulsan, 680-749 South Korea (e-mail: hmcho67@mail.ulsan.ac.kr).

Sang-Bock Cho, Professor, is with the School of EE, University of Ulsan, Ulsan, 680-749 South Korea (e-mail: sbcho@ulsan.ac.kr).

In recent years, general purpose processors have been endowed with functional units of Single Instruction Stream Multiple Data Stream (SIMD) operation [5]. The present rapid growth of CPU power available in a personal computer will allow real-time execution of motion-related tasks in software on a general CPU.

In this paper, we present the performance enhancement of motion estimation algorithms by using the Streaming SIMD Extensions 2 (SSE2) with Intel Pentium 4 processors. SSE2 technology is designed to accelerate performance of applications involving floating-point based code and algorithms that operate on blocks of data. Our goal is to reduce the CPU executed time for running motion estimation algorithms. Using the new packed byte data type, eight pixel bytes can be simultaneously executed at once instead of executing each byte one at a time for eight cycles. Hence, the speed-up in performing algorithms is achieved. Two algorithms used in our experiment are Full Search Algorithm and Three Step Search Algorithm as motion estimation processing.

II. Review Of Motion Estimation Algorithms

The basic idea in video compression is to eliminate spatial, temporal and statistical redundancy. Motion estimation is the process of calculating motion vectors by finding matching blocks in the current frame corresponding to blocks in the reference frame to eliminate the spatial and temporal redundancy. And this plays an important role in inter-frame predictive coding system. The reducing computational complexity of motion estimation in video compression is strongly requested because the sub-pixel unit motion estimation operates to obtain high compression ratio in recent video compression algorithm such as MPEG-4 and H.264. Therefore, motion estimation is the main power consuming block in video compression technology. Various search algorithms have been announced for estimating motion.

A. Full Search Algorithm

Full search is an exhaustive search algorithm and it is the simplest method to find the motion vectors for each block. In it, the mean of absolute difference (MAD) is found at each point (i, j) in the search region [2].

For each motion vector with the search region p, the mean of absolute difference function has to evaluate $(2p+1)^2$ times for each macro-block. At each searching point (i, j), we compare $M \times N$ pixels and each pixel comparison requires three operations, namely: a subtraction, an absolute value calculation and an addiction. Thus the total complexity per block is $(2p+1)^2 \times MN \times 3$ operations. For an F frame rate with $I \times J$ image resolution, the overall computational complexity is $(IJF) \times (2p+1)^2 \times 3$ operations per second.

This makes it as a very computationally intensive method. CPU executed time for Full Search is the highest of all the motion estimation algorithms. At the same time, the accuracy of Full Search algorithm is also highest and the best match for every block in the current frame is always found. Full Search algorithm, therefore is a benchmark for comparison of the quality of a search algorithm. There is a trade-off relationship between the efficiency of the algorithms and the quality of the prediction image. Keeping this trade-off in mind, a lot of algorithms have been developed.

B. Three Step Search Algorithm

Three-step Search algorithm is one of the fast search algorithms to reduce high computation complexity. Three-step Search algorithm has been widely used in block matching motion estimation due to its simplicity and effectiveness [2]. It is very suitable for searching large motion and finding minimum globe especially for those sequences with large motion base on the sparsely distributed checking points in the first step. For the search region $p = 7$, the total number of computations for searching are fixed as 25 points in comparison with 289 points of Full Search algorithm. Hence, the CPU executed time for Three-step Search algorithm is less than Full Search algorithm at acceptable quality of image resolutions.

III. Single Instruction Multiple Data (SIMD) Architecture

Usually, a processor processes one data element in one instruction and that processing style is called Single Instruction Single Data, or SISD. In contrast, a processor with the Single Instruction Stream Multiple Data Stream or SIMD capability processes multiple data elements in one instruction [5]. The Single Instruction Stream Multiple Data Stream (SIMD) Architecture performs the operations on many elements in a lockstep fashion. The same instruction is performed on different data elements which are computed by differently functional units. The Intel's MMX/SSE/SSE2, AMD's 3DNow, and Power PC's Altivec ISA extensions are testimonial to the benefits of SIMD support to traditional superscalar processors.

A. Intel's Streaming SIMD Extension

SIMD Extensions for the IA-32 ISA began with the Multimedia Extensions (MMX) in 1997 for the Pentium processor. MMX data type contains of 64 bits sub-word parallel ALU's for byte, word, double word and quad word that enhances its performance on multimedia benchmarks.

However, these instructions had a strongly limited function, in that only integer data types could be handled. Also, since the MMX instructions utilized the floating point registers, it was very hard to inter-mingle floating point and MMX instructions [5].

Streaming SIMD Extensions (SSE) from the Intel Pentium III marked the advent of 70 new instructions to the IA-32 ISA. The biggest winners from the new instructions were applications that handled 3D or streaming media. Applying identical instructions to multiple pieces of code for these applications was now handled in parallel.

The SSE2 technology from the Intel Pentium-4, introduced new SIMD double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel architecture [6]. The 128-bit SIMD integer extensions are a full super-set of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, convert between integer and floating-point data types, and make efficient operations between the caches and system memory [4]. These instructions provide a mean to accelerate operations for typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application.

B. SSE versus MMX

MMX and SSE, both of which are extensions to existing architectures, share the concept of SIMD, but they differ in the data types that they handle, and in the way they are supported in the processor.

MMX instructions are SIMD for integers, while SSE instructions are SIMD for single-precision floating-point numbers. MMX instructions operate on two 32-bit integers simultaneously, while SSE instructions operate on four 32-bit floats simultaneously.

The main difference between MMX and SSE is that no new registers have been defined for MMX, while eight new registers have been defined for SSE. Each of the registers for SSE is 128 bits long and can hold four single-precision floating-point numbers (each being 32 bits long) [6].

The MMX registers have been allocated out of the floating-point registers of the floating-point unit. A floating-point register is 80 bits long, of which 64 bits are used for an MMX register [5]. A limitation of this architecture is that an application cannot execute MMX instructions and perform floating-point operations simultaneously. Additionally, a large number of processor clock cycles are needed to change the state of executing MMX instructions to the state of executing floating-point operations and vice versa. SSE does not have such a restriction. Separated registers have been defined for SSE. Hence, applications can execute SIMD integer (MMX) and SIMD floating-point (SSE) instructions simultaneously. Applications can also execute non-SIMD floating-point and SIMD floating-point instructions simultaneously.

The arrangement of the registers in MMX and SSE is illustrated in Figure 1.

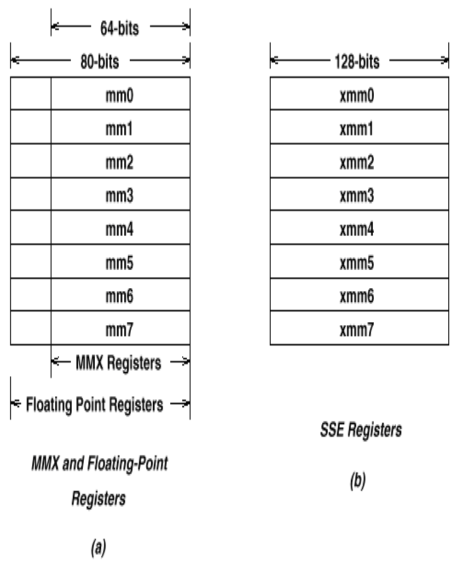

Fig. 1: MMX and SSE registers

IV. Experiment And Result

We performed motion estimation using Full Search and Three-step Search algorithm for both 16 × 16 and 8 × 8 block size of images. All experiments were run on a Pentium 4 processor and the results were compared between ordinary C code and code using SSE2 technology. The performance between two algorithms without using SSE2 technology was also compared.

There was limited support compiler available for the SIMD ISA extension. As a result to make use of the rich features provided by this extension, we needed to go through different programming techniques. One of the following techniques could be used to code programs with SSE2:

- Assembly level programming
- Intrinsics
- Vector Class Library

A. *SSE2 Coding Technique*

We used Intrinsics and Vector Class Library for our programs. The advantage of using Intrinsics and Vector Class Library is that the Intrinsics and Vector Classes free the programmer from managing registers while ensuring easier maintenance and modularization of code. The compiler optimizes instruction scheduling and register allocation to make the executable programs run faster.

Each computation and data manipulation assembly instruction has a corresponding intrinsic that implements it directly. The intrinsics in SSE2 contain suffixes to indicate which data type is operated on by instructions.

- p, pd, ps suffix indicates a packed, packed double, packed single precision floating-point operation.
- s, sd, ss suffix indicates a scalar, scalar double or scalar single precision floating-point operation.
- i, si, su suffix indicates an integer, 64-bit signed or unsigned integer.
- pi, pu, epi, epu suffix indicates 128-bit signed or unsigned integer extended precision operation for 8, 16, 32 or 64 bits.

To use the intrinsics library, the file xmmintrin.h must be included into the programs. Thus we chose to utilize the Intrinsics style of coding for our motion estimation programs. The Intel's C++ compiler was chosen to compile our programs. For most of the parts in our programs, normal C code constructs were used.

B. *Code Snippet*

Blockdiff is the main computationally intensive function called in the program. It can also make use of the SSE2 technology to improve its performance. We made the code using intrinsics to employ the SSE2 data path.

Snippet below provides the blockdiff function call for processing 16 x 16 block size:

```
   int blockdiff(int x1, int x2, int  y1, int  y2)
{
   unsigned char block1[16], block2[16];
int i1, j1, k1, ch, offset1, offset2;
int diff1[16][16],  totaldiff = 0;
FILE *fp1, *fp2;
__m128i *b1,*b2, m1;

union sse2
{
  __m128i m;
}m;
...
...
// type casting pointers.
b1 = (__m128i*) block1;
b2 = (__m128i*) block2;
// SAD for 16 bytes.
m1 =_mm_sad_epu8(*b1,*b2);
m.m = m1;
...
...
}
```

The snippet shows the SSE2 code for the blockdiff function which finds the differences between two blocks located at (x1, y1) and (x2, y2).

The top part shows the declarations inside the function and the bottom part shows the calculation of the differences using the SSE2 intrinsic. We defined a union called sse2, which could be used to address the m register of the sse2 data type "__m128i" and as an array of 8 integers as well. This __m128i register consists of 16 8-bit integer values.

The block1 and block2 arrays will contain the 16 8-bit pixel values from the image. These are typecast into the __m128i format and put into the locations pointed by b1 and b2. Next the _mm_sad_epu8() instruction will find the sum of differences of these 16 values directly and will place it in the m1 register.

The following snippet provides the blockdiff function call for processing 8 x 8 block size:

```
   int blockdiff(int x1, int x2, int  y1, int  y2)
{
   unsigned char block1[8], block2[8];
int i1, j1, k1, ch, offset1, offset2;
int diff1[8][8],  totaldiff = 0;
FILE *fp1, *fp2;
__m64 *b1,*b2, m1;

union sse2
{
  __m64 m;
}m;
...
...
// type casting pointers.
b1 = (__m64*) block1;
b2 = (__m64*) block2;
// SAD for 16 bytes.
m1 =_m_psadbw(*b1,*b2);
m.m = m1;
...
...
}
```

This blockdiff function code for taking differences between 8 x 8 blocks is similar to the one above. The operations of 8 x 8 blocks were performed similar to the 16 x 16 blocks but the data type and intrinsic used were different. The data type used was the __m64 type which consisted of 8 8-bit values. The intrinsic used to calculate the sum of differences was the _m_psadbw() operation.

C. Result

TABLE I
PERFORMANCE COMPARISON FOR 16 X 16 BLOCK SIZE

Algorithm	Without SSE2	With SSE2	Speed-up
Full search	25s	6s	4.1
Three-step search	7s	2s	3.5

TABLE II
PERFORMANCE COMPARISON FOR 8 X 8 BLOCK SIZE

Algorithm	Without SSE2	With SSE2	Speed-up
Full search	29s	8s	3.6
Three-step search	12s	4s	3.0

We draw the following summary from the results given above. The CPU executed time for Three-step Search algorithm costs much less than the one for Full Search algorithm. We notice that the speed-up is by a factor of 3.0 - 4.0 for most programs using SSE2 technology. The 8 x 8 block size programs for both algorithms took longer to execute in comparison with the 16 x 16 block size programs. The reason is that the loop overhead for the programs goes up even though the number of additions or subtractions to be performed are the same. The 8 x 8 blocks perform a better job at matching than the 16 x 16 blocks. Moreover, the 8 x 8 block size programs are better suited for tracking movement of the smaller image regions with these algorithms.

V. CONCLUSION

SIMD extensions to the superscalar architectures have helped to improve the performance of general purpose processors on media applications. In this paper, we show that it is possible to reduce the time required for the algorithm significantly by writing code for one specific processor and exploit all its capabilities. Experiment results show that for both 16 x 16 block size and 8 x 8 block size, the algorithms proposed with SSE2 technology performed better than the algorithms without using SSE2 technology. The programs with SSE2 technology actually reduce the CPU executed time 3 - 4 times while the image resolution is kept at the same quality. Therefore, SIMD architecture is suitable for multimedia applications.

REFERENCES

[1] T. Wiegand, G. J. Sullivan, G. Bjontegaard and A. Luthra, “Overview of the H.264/AVC Video Coding Standard,” *IEEE Trans. Circuits. Syst. Video Technol.*, vol. 13, pp. 560–576, July 2003.

[2] X. Jing and L. P. Chau, “An Efficient Three-Step Search Algorithm for Block Motion Estimation,” *IEEE Trans. Mutimedia*, vol. 6, pp. 435–438, June 2004.

[3] D. Talla, L. K. John and D. Burger, “Bottenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements,” *IEEE Trans. Computers*, vol. 52, pp. 1015–1031, August 2003.

[4] “Using SSE2 in Motion Compensation for Video Decoding and Encoding,” Intel Application Note, AP-942.

[5] A. Peleg and U. Weiser, “MMX Technology Extension to the Intel Architecture,” *IEEE Micro*, August 1996.

[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, “The Microarchitecture of the Pentium 4 Processor”, *Intel Technology Journal Q1*, 2001.