Block-Matching In Motion Estimation Algorithms Using Streaming SIMD Extensions 2 (SSE2)

Version 2.0

7/00

Order Number 248605-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Pentium® 4 processor update	7/00
1.0	Original publication of document	9/99

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. Digital Television, Benoit, H. Arnold Publishing Co., 1997.
2. MPEG Video Compression Standard, Mitchell, Joan L. et al., Chapman and Hall Publishing, 1995.

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) technology introduces 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 the Streaming SIMD Extensions (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.

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). 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;
```

This paper discusses only the more commonly used SAD algorithm.

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.

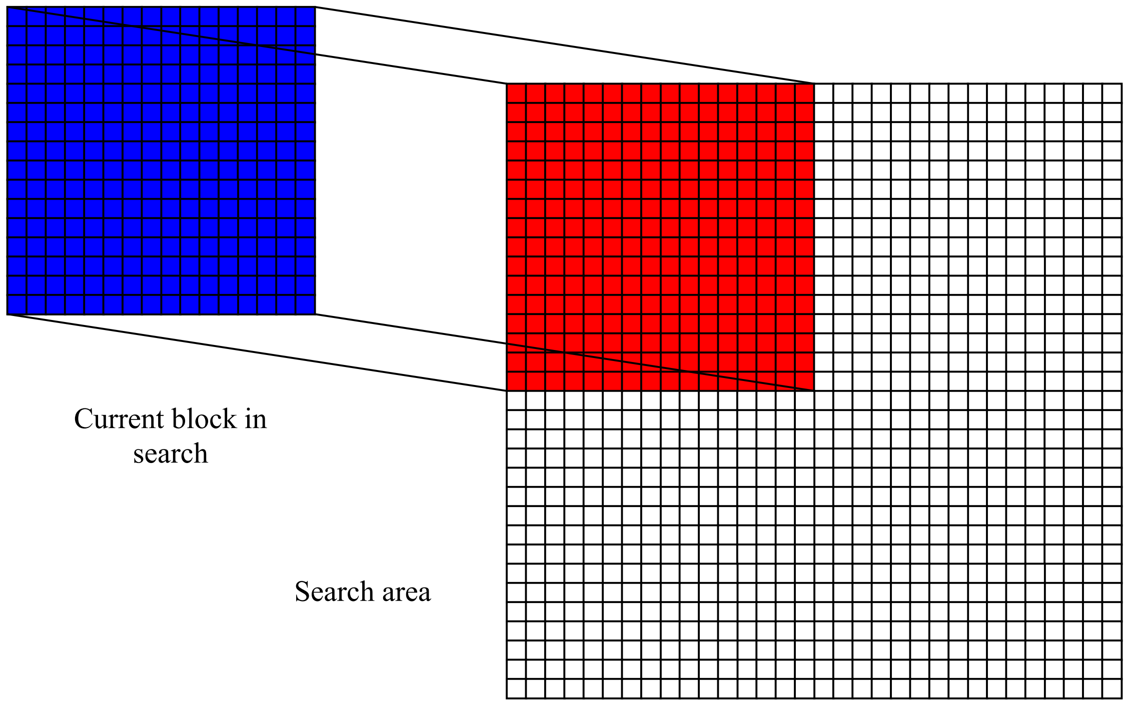

Figure 1: Comparing the First Set of Blocks

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.

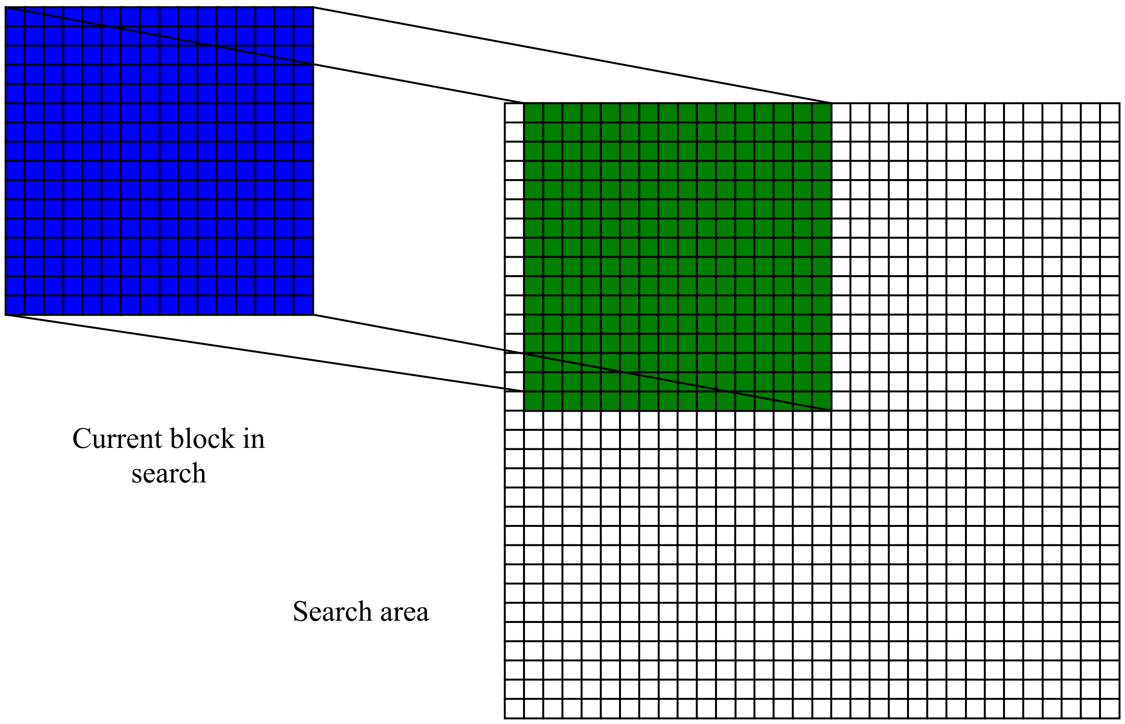

Figure 2: Comparing the Next Blocks

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.

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 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.

3 Performance

3.1 Gains/Improvements

Use of the 128-bit version of the `psadbw` instruction yielded a significant performance increase over the SSE version of the code, which used the 64-bit version of psadbw. The size of the functions was also dramatically cut, by nearly 50%, since only one SAD operation per row was required.

`Psadbw` returns two separate SAD values, one for the lower eight bytes and one for the upper eight bytes, which must be added together to obtain the final value. Note that this needs to be done only once per block-match operation, not once per row.

The total performance improvement for the Pentium 4 processor versus a Pentium III processor at 733 MHz was also excellent. This improvement was due to three main factors:

- increased memory bandwidth (3.2GB/sec vs. 800MB/sec on the Pentium III processor) allowing more data to be loaded quickly from external memory
- improved scheduling of uops
- improved branch prediction on the outer search loops, due to having more bits available for tracking when counted loops end

The benefits of the new architecture are greater than those resulting just from using the new 128-bit instructions.

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 see data cache splits. These occur when part of the data we wish to load is contained in each of two different cache lines. In previous generations of Intel processors, it was sometimes faster to load the two cache lines and then manually construct the data than to endure the high penalty. Using the `movdqu` instruction on the Pentium 4 processor is faster than to perform the same operation algorithmically.

3.2 Considerations

Theoretically, it should be possible to achieve a 2X speedup by using the new double width `psadbw` instruction and proper instruction pipelining. This is limited, however, by the need to load data which is not 16-byte aligned. All implicit loads and the movdqa instruction require 16-byte alignment or they will fault. Referring to the diagrams describing the motion estimation search, you may notice that we are required to move the search one pixel at a time through the entire search area; obviously, we cannot be 16-byte aligned for each of those loads!

This same behavior, moving one pixel at a time across the array, causes data cache line splits. This is the situation where one load requires data from 2 cache lines. This is particularly slow, since it requires that two loads be done and then an extra operation to combine the pieces into the requested data. To clarify the concept, picture doing an integer (32bit/4byte) load that begins with an address at the last byte of a cache line; one byte will come from the first cache line and 3 bytes will come from the second cache line. These two pieces must be assembled internally by the processor before becoming available for calculations.

The instruction set provides a second, unaligned load instruction, `movdqu`. This instruction can handle any type of misalignment, including data cache splits, at the cost of speed. It is implemented as a series of uops that create aligned data. Use of this instruction decreases the best possible speed gain.

One important scheduling issue is the handling of the `psadbw` and `paddw` instructions. Both instructions take several cycles to execute, and there is no overlap in execution time between two `psadbw` or between two `paddw` instructions scheduled consecutively in the code. If a `psadbw` and a `paddw` instruction are executed consecutively, however, they can begin execution on consecutive clocks, allowing both instructions to be executed in an overlapping manner. The net result is high performance in the calculation portion of the code.4 Conclusion

The SSE2 instructions have been shown to give a significant gain in performance on the most processor intensive part of MPEG encoders, while reducing code size and complexity. Due to limitations in the specifications of the motion estimation search, we are unable to reach the theoretical best performance that the architecture could achieve.

Overall, the Pentium 4 processor architecture is a boon to this algorithm. If a typical MPEG encoder uses 50% of the CPU for motion estimation searches, then this could be reduced to a minor percentage on a Pentium 4 processor. Alternately, a much more accurate search could be done without increasing the execution time, while increasing the quality of the encoded video.

4 Conclusion

The SSE2 instructions have been shown to give a significant gain in performance on the most processor intensive part of MPEG encoders, while reducing code size and complexity. Due to limitations in the specifications of the motion estimation search, we are unable to reach the theoretical best performance that the architecture could achieve.

Overall, the Pentium 4 processor architecture is a boon to this algorithm. If a typical MPEG encoder uses 50% of the CPU for motion estimation searches, then this could be reduced to a minor percentage on a Pentium 4 processor. Alternately, a much more accurate search could be done without increasing the execution time, while increasing the quality of the encoded video.

5 C/C++ Coding Example

```
// These outer loops control the movement through the search range
// pucC points to the current (stationary) block, pucR
// points to the block in the search area.
// iTop, iBottom, iLeft, iRight are the edges of the search range.
// iWidth is the number of pixels across a frame.
for(iY = iTop; iY <= iBottom; iY++){
        pucRefLeft = pucRef + (iY * iWidth);   // Setup pointer into search
                                              //  range for the current Y.
        for(iX = iLeft; iX <= iRight; iX++){
          iTmpAd = 0;                         // Initialize accumulator
          pucC = pucCurLeftTop;               //  Set pointers for current X
          pucR = pucRefLeft + iX;

          // These inner loops control the movement across the block
         //  as we do the SAD operation pixel by pixel.
          for(iV = 0; iV < 16; iV++){      // For each pixel in 16X16 block,
            for(iH = 0; iH < 16; iH++){   // calculate the abs difference
              iTmpAd += abs((ME_INT32)*(pucC++) - (ME_INT32)*(pucR++));
            }
            pucC += iDown;  // Move pointers to next row of block
            pucR += iDown;
          }

          /* Check for a new minimum SAD. If it is a new min, store
              position information */
          if(iMinAd > iTmpAd){
            iMinAd    = iTmpAd;
            *piMvPos    = iX;
            *(piMvPos+1) = iY;
          }
        }
      }
      return iMinAd;     // Return best SAD value from search
```

6 SSE2 Intrinsics Code Example

```
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_loadu_si128((__m128i *)pucR);
   row4 = _mm_loadu_si128((__m128i *)(pucR + iWidth));
   row6 = _mm_loadu_si128((__m128i *)(pucR + 2*iWidth));
   row8 = _mm_loadu_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);
   row8 = _mm_sad_epu8(row7, row8);
   sum = _mm_add_epi16(sum, row5);
   sum2 = _mm_add_epi16(sum2, row7);


   row2 = _mm_loadu_si128((__m128i *)(pucR + 4*iWidth));
   row4 = _mm_loadu_si128((__m128i *)(pucR + 5*iWidth));
   row6 = _mm_loadu_si128((__m128i *)(pucR + 6*iWidth));
```

```
row8 = _mm_loadu_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_loadu_si128((__m128i *)(pucR + 8*iWidth));
row4 = _mm_loadu_si128((__m128i *)(pucR + 9*iWidth));
row6 = _mm_loadu_si128((__m128i *)(pucR + 10*iWidth));
row8 = _mm_loadu_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, row1);
sum2 = _mm_add_epi16(sum2, row3);


row2 = _mm_loadu_si128((__m128i *)(pucR + 12*iWidth));
row4 = _mm_loadu_si128((__m128i *)(pucR + 13*iWidth));
row6 = _mm_loadu_si128((__m128i *)(pucR + 14*iWidth));
```

```
      row8 = _mm_loadu_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);
      sum = _mm_add_epi16(sum, row1);
      sum2 = _mm_add_epi16(sum2, row3);
      sum = _mm_add_epi16(sum, sum2);

      tmp = sum;
      sum = _mm_srli_si128(sum, 8);
      sum = _mm_add_epi32(sum, tmp);

      // Check for new minimum AD
      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

Revision	Revision History	Date
2.0	Updated with 1.2 GHz Pentium 4 processor performance data	7/00
1.0	Original publication of document	9/99

Table 1: Performance Data of ME Implementations

Performance Data in Milliseconds		
	Pentium III Processor (733 MHz)	**Pentium 4 Processor (1.2 GHz)**
Streaming SIMD Extensions (SSE)	25.6	13.6
Streaming SIMD Extensions 2 (SSE2)	-	10.8

Table 2: Speedups from Table 1 Performance Data

Implementations and Platforms	Speedup
Pentium 4 processor (SSE2 vs. SSE)	1.26
SSE (Pentium 4 processor vs. Pentium III processor)	1.89
SSE2 on Pentium 4 processor vs. SSE on Pentium III processor	2.38

Performance was measured using a Pentium III 733 MHz processor and a Pentium 4 1.2 GHz processor. See Test Systems Configuration on page A-2 for a detailed description of the test systems.

As can be seen in Table 2, the SSE2 implementation is 1.26 times faster than the SSE implementation when both implementations are executed on a Pentium 4 processor. This speedup is attributed to the following optimizations:

- Larger SIMD width. The integer SSE2 use the 128-bit XMM registers instead of the 64-bit MMX™ technology registers. The increased SIMD width allows an entire row of a macroblock to be compared with a single instruction.
- The decreased number of instructions due to the wider `psadbw` and load instructions decreased the code size by nearly 50%.
- The increased memory bandwidth of the processor allowed more data to be loaded in a given amount of time. Since this algorithm thrashes the caches frequently, this gave a significant speedup.

- The speedup was limited by the use of one unaligned load for each row. This is required since the reference frame data will not be aligned, as the search moves across the frame on a pixel by pixel basis. This limited the best theoretical speedup to 1.33X.

Test Systems Configuration

Table 3: Pentium III Configuration

Processor	**Pentium III Processor at 733 MHz**
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 4: Pentium 4 Configuration

Processor	**Pentium 4 Processor at 1.2 GHz**
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195