White Paper

Using Intel® Advanced Vector Extensions to Implement an Inverse Discrete Cosine Transform

Intel® Software & Services Group

Richard Hubbard

September 2010

Intel Corporation

Introduction

Transform coding is an important step of image and video processing applications. Pixels in an image have a level of correlation with their neighboring pixels. Adjacent pixels in successive frames show a very high correlation. These correlations can be used to predict the value of a pixel from its neighbor. The highly correlated spatial data is transformed into uncorrelated coefficients in the frequency domain. The transformed data, represented as coefficients, are independent, allowing them to be manipulated separately. The human eye is more perceptive to low-frequency changes than high-frequency changes. Encoders can achieve compression after setting the high-frequency coefficients to zero. An inverse transform is used in the decoder pipeline to reconstruct the source data.

Figure 1 and Figure 2 depict the components of a typical image encoder/decoder system.

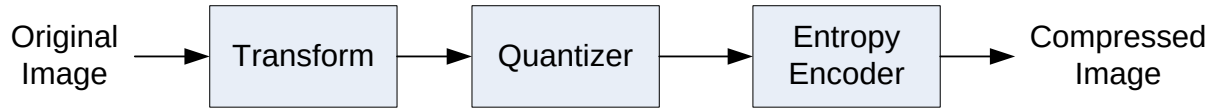

Figure 1 - Source Encoder Block Diagram

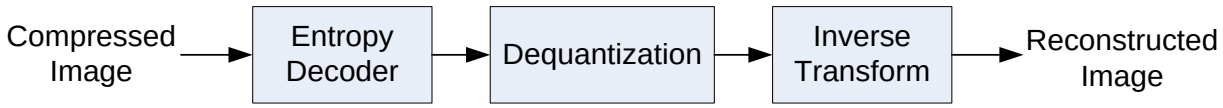

Figure 2 - Source Decoder Block Diagram

The Discrete Cosine Transform (DCT) and Inverse Discrete Cosine Transform (IDCT) are widely used in the encoder and decoder pipelines of image processing systems such as MPEG, JPEG, and other standards. A fast and accurate IDCT transformation is crucial for the best user experience.

Intel® Advanced Vector Extensions (Intel® AVX), introduced with the new Intel® processor microarchitecture codenamed Sandy Bridge, extends the capabilities of Intel® Streaming SIMD Extensions (Intel® SSE) for floating point data and operations such as Inverse Discrete Cosine Transforms. Intel® AVX essentially doubles the width of the current XMM registers and adds new extensions that can operate on the wider data width. Intel® AVX significantly increases the floating-point performance density with improved power efficiency over previous 128-bit SIMD instruction set extensions. This document specifically examines how Intel® AVX and Sandy Bridge microarchitecture features such as wider 256-bit SIMD registers, non-destructive source operands, new data manipulation and arithmetic primitives, dual 128-bit load ports, and doubling of computational execution width can benefit the Inverse Discrete Cosine Transform (IDCT) operation. Intel® AVX improves the computational throughput of high performance precision-sensitive floating point transform applications. Integer implementations may also benefit from features such as non-destructive source operands.

Intel also provides a set of Intel® AVX software development tools like the Intel® AVX-enabled Intel® C++ Compiler, the Intel® Software Development Emulator (SDE), and the Intel® Architecture Code Analyzer. All of these tools were effectively used during the development of this kernel, and can be downloaded from the Intel® AVX website: http://software.intel.com/en-us/avx/

Testing Environment

The performance speedups stated in this paper are based on runs on actual Sandy Bridge microarchitecture-based pre-production silicon. It can also be assumed that the test data is already in the first level processor cache prior to the computation of the IDCT algorithm. Performance comparisons are made based on the relative performance of Intel® AVX versus corresponding Intel® SSE implementations using C intrinsic instructions, both run on the Sandy Bridge microarchitecture-based silicon. The code was compiled using the 64-bit Intel® C++ Compiler, version 11.1.038. The applications were compiled using the following command line options:

- Intel® SSE: /QxSSE4.1 /O3
- Intel® AVX: /QxAVX /O3

A comparison of short integer performance was performed by compiling the Intel® SSE implementation for different architectures. The compiler will generate VEX-encoded Intel® SSE instructions when the /QxAVX switch is specified. The compiler will generate Intel® SSE 4.1 instructions when the /QxSSE4.1 switch is specified. Non-destructive source operands are introduced with the VEX-encoded instructions.

The test application follows the 8x8 IDCT accuracy requirements as documented in section 3 of the IEEE standard 1180-1900 (Reference 5). Randomly generated input data is processed by a reference DCT. The output of the DCT is rounded to the nearest integer and clipped such that -2048 <= value < 2047. The clipped values are inputs to both the reference IDCT and the IDCT under test. Their outputs are rounded to the nearest integer and clipped such that -256 <= output < 255. The two results are then compared and accuracy measurements are made.

In this application, the references are implemented using double precision floating point scalar C code. Several proposed IDCT implementations are tested; a short integer Intel® SSE version, a single precision floating point Intel® SSE version, and a single precision floating point Intel® AVX version. All the vector versions are implemented with C intrinsics.

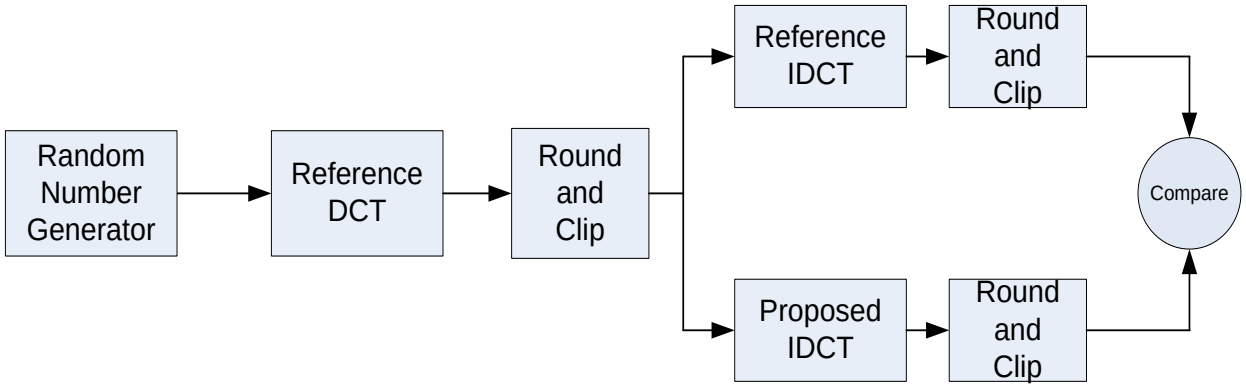

Figure 3 - Test Application

Discrete Cosine Transforms

The Discrete Cosine Transform (DCT) converts the spatial data of an image into the frequency domain. The mathematical operations are described in detail in Reference [1].

The two-dimensional (2D) DCT transforms 64 pixel values, an 8x8 block of pixels, producing 64 coefficients. The image's pixel values are transformed into coefficients derived from amplitudes of cosine basis functions.

Equation 1 - 1D DCT

$$F(u) = \frac{1}{2}C(u)\left[\sum_{x=0}^{7} f(x)\left(\cos\frac{(2x+1)u\pi}{16}\right)\right]$$

Equation 2 - 2D DCT

$$F(v,u) = \frac{1}{4}C(v)C(u)\left[\sum_{y=0}^{7}\sum_{x=0}^{7} f(y,x)\left(\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right)\right]$$

Equation 3 - 1D IDCT

$$f(x) = \sum_{u=0}^{7}\left(\frac{C(u)}{2}F(u)\cos\frac{(2x+1)u\pi}{16}\right)$$

Equation 4 - 2D IDCT

$$F(v,u) = \sum_{v=0}^{7} \frac{C(v)}{2} \sum_{u=0}^{7} \frac{C(u)}{2} F(v,u) \left[\cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right]$$

Where:

C(u) = 1/√2 for u = 0	C(v) = 1/√2 for v = 0
C(u) = 1 for u > 0	C(v) = 1 for v > 0
f(x) = 1D sample value	f(y, x) = 2D sample value
F(u) = 1D DCT coefficient	F(v, u) = 2D DCT coefficient
x, y	Coordinates in the non-transformed domain
u, v	Coordinates in the transformed domain

Reference work [2] summarizes the number of mathematical operations; the numbers are worth repeating. Comparing the DCT and IDCT equations, it can be shown that the transforms require the same number of operations. An analysis of Equation 2 shows that the 2D DCT requires 64 multiplications and 63 additions for each coefficient. Therefore 4096 multiplications and 4023 additions are required to transform an 8x8 block. Replacing the 2D DCT with 8 1D DCTs for the 8 rows and 8 1D DCTs for the 8 columns can reduce the number of operations. Equation 1 shows that the 1D DCT requires 64 multiplications and 56 additions producing 8 coefficients. Transforming the 8x8 block with 1D DCTs requires 1024 multiplications and 896 additions. References [1] and [4] discuss other DCT and IDCT algorithms.

Previous Work and Specifications

The following previous works and specifications should be consulted for background information:

- Reference [1] for the JPEG specification.
- The IDCT algorithm used in the Intel® AVX implementation discussed in this whitepaper follows the optimized algorithm discussed in Reference [2].
- Reference [3] provides additional background information.

Intel® AVX Implementation of IDCT

As in the previous works, the algorithm in this whitepaper performs 8 1D IDCT transforms on the rows of an 8x8 block. Then 8 1D IDCT is performed on the columns of those results. Two different 1D IDCT transforms are used, and neither transform requires a transpose.

The AVX implementation operates on two rows of data simultaneously. It begins by loading four floats from each row into separate 128-bit registers. The reason for performing 128-bit loads will become clearer in a moment. It is important to remember that the 128-bit XMM registers overlay the lower 128-bits of the corresponding 256-bit YMM register. The upper 128-bits of the YMM register is loaded with the next set of four floats from those rows via the _mm256_insertf128_ps instruction. The resultant YMM register contains the first set of four floats from one row in the lower 128-bits, and the second set of four floats from the second row in the upper 128-bits. This seems like a lot of work to load eight floats, so why take this approach? This is known as a *strided load* (because the next load will read from an area of memory that is more than a unit stride away from the current memory location).

The strided load is effective in certain applications.

- Eight products need to be summed in the IDCT and it is more straightforward to sum the products if they are in the same 128-bit lane of two YMM registers. Two registers are built with the contents from one row in the bottom 128-bits of both registers, and the contents from the second row in the upper 128-bits of both register. This is depicted in Figure 4 through Figure 6.

- Another benefit of the strided load approach is that data movement, or placement of the 128-bit memory operand into the upper 128 bits, can take place on a different execution port than the shuffle port. This reduces the utilization of the port that performs shuffles, and allocates the work more evenly across the execution ports.

- One last benefit of placing floats 4-7 of two rows into one register allows for a reduction in shuffles that takes place near the end of row processing.

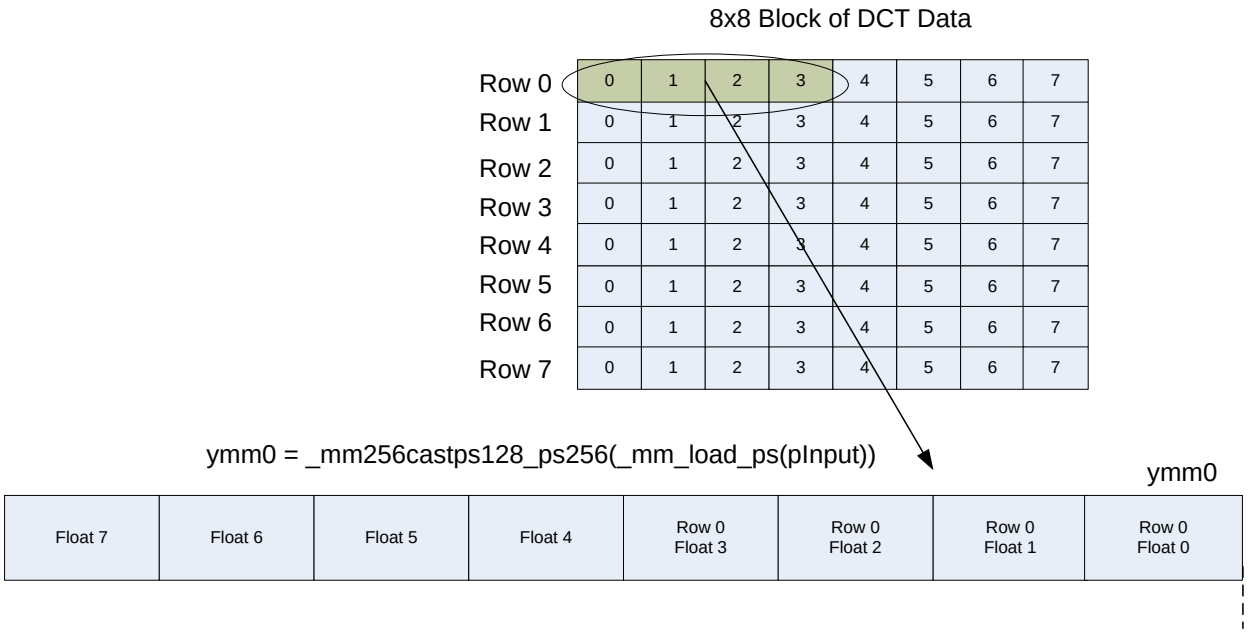

Figure 4 - Loading Four Floats from Row 0

The combination of the *_mm_load_ps* and *_mm256_castps128_ps256* instructions loads 128-bits into the least significant 128-bits of a ymm register.

Figure 5 depicts the strided load of Row 4 data, and the insertion into the most significant 128-bits of ymm0.

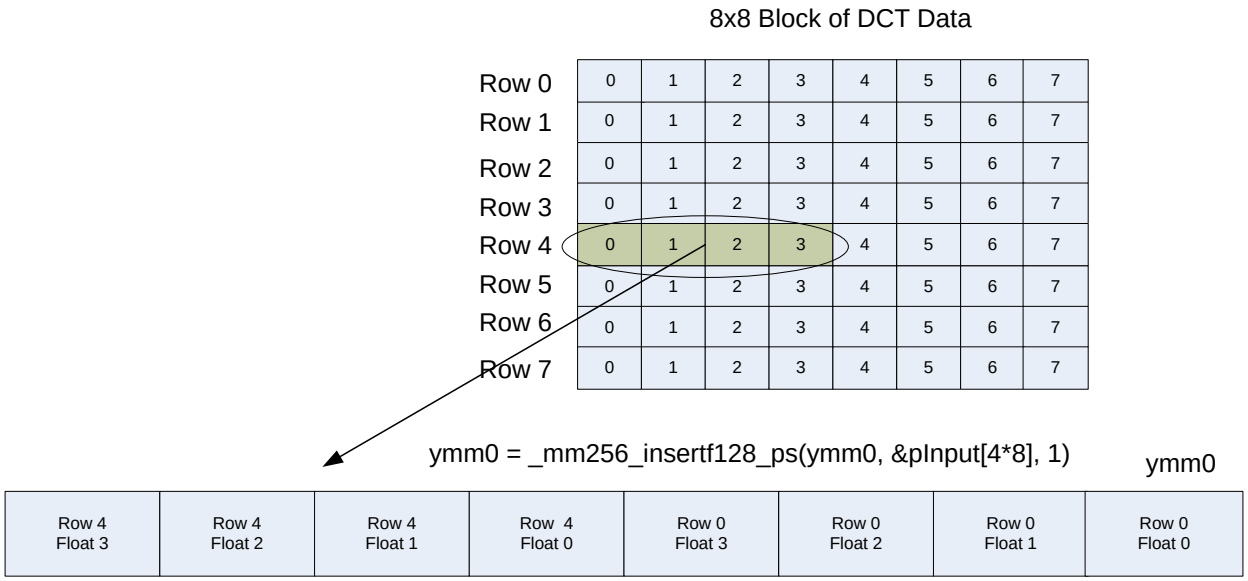

Figure 5 - Strided Load of Second Row

A second YMM register is populated in a similar manner; it contains the second set of four floats (floats 4-7) from both rows. The results are shown in Figure 6.

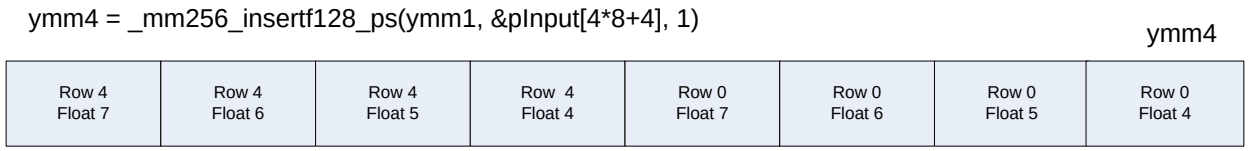

Figure 6 - Second Set of Floats from Two Rows

The input data must be multiplied by different cosine terms. A single float from each row is broadcasted via the *_mm256_shuffle_ps* to prepare for the multiplication, as shown in Figure 7. In this case float 0 is duplicated.

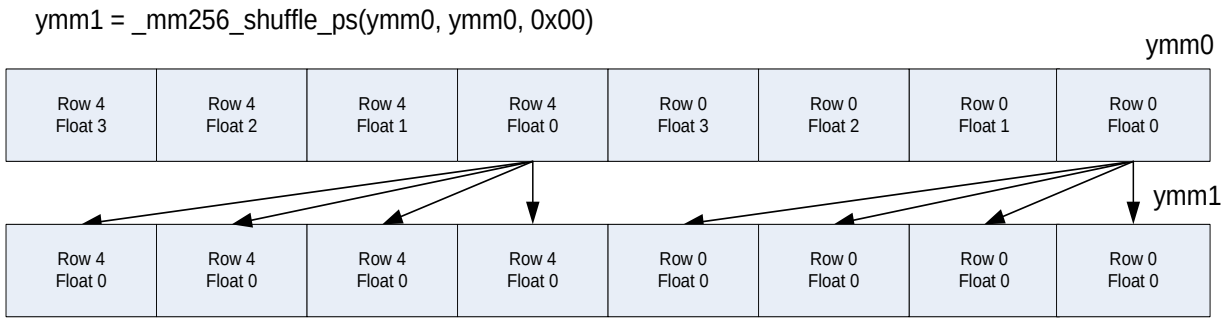

Figure 7 - Preparing for Cosine Multiplication

Two *_mm256_mul_*ps instructions perform the necessary cosine multiplications. This completes one of the seven sums of product terms.

This sequence of instructions is repeated three times, operating on a different pair of input values and cosine terms each time. At this point there are four registers

containing sums of products. One register contains sums of products for floats 0 and 2, another contains sums of products for floats 1 and 3, 4 and 6, and finally 5 and 7.

Three more sums are necessary to complete the sum of products. The sums of products of all even terms are added, and the sums of products of all odd terms are added. That is only two of the three sums.

```
ymm_even = _mm256_add_ps(r_ymm02, r_ymm46);
ymm_odd = _mm256_add_ps(r_ymm13, r_ymm57);
```

The seventh and final sum produces the least significant four floats of the output. A subtraction is necessary to produce the most significant four floats; however, they are not in the correct order.

```
ymm_diff = _mm256_sub_ps(ymm_even, ymm_odd);
ymm_sum = _mm256_add_ps(ymm_even, ymm_odd);
```

The order of the most significant floats of two rows can be properly adjusted using a single *_mm256_shuffle*_ps instruction, as shown in Figure 8. The benefit of placing the most significant four floats from each input row is more apparent.

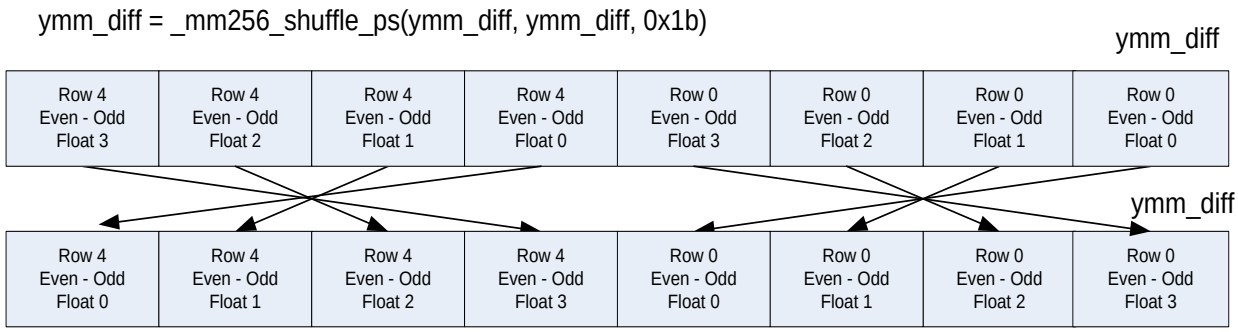

Figure 8 - Reversing the Order of Differences

The final step for the 1D IDCT for these two rows is to recombine the results for each row into the same 256-bit register using two *_mm256_ permute2f128* instructions.

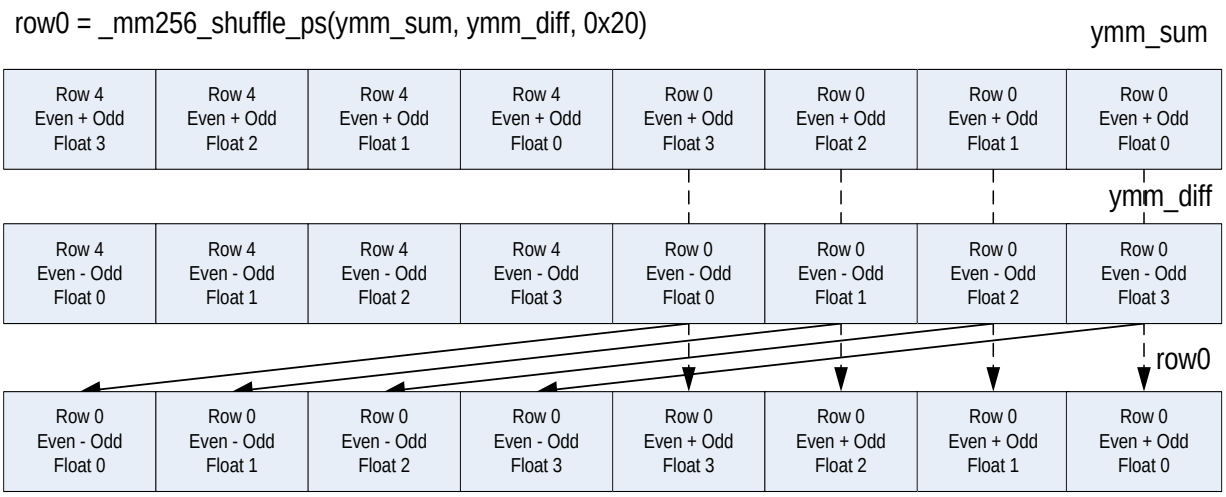

Figure 9 - Recombining Row 0

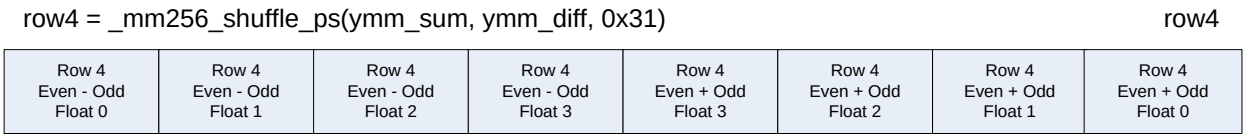

Figure 10 - Recombining Row 4

That completes the 1D IDCT for two sets of rows. The sequence is repeated for rows 1 and 7, 2 and 6, and finally 3 and 5 to complete the 1D IDCT for eight rows.

Now that the 1D IDCT of the eight rows is completed, the 1D IDCT of the eight columns can begin. The algorithm is similar, yet simpler because the sums of products are already in the proper position for the addition. The 1D IDCT of the columns does not require any shuffles.

It was possible to eliminate four add instructions in the floating point IDCT of the columns.

```
//row5*tangent + row5
r_ymm0 = _mm256_mul_ps(row5, tangent);
r_ymm0 = _mm256_add_ps(row5, r_ymm0);

//row3*tangent + row3
r_ymm1 = _mm256_mul_ps(row3, tangent);
r_ymm1 = _mm256_add_ps(row3, r_ymm0);
```

A single multiply of the row times the tangent term plus 1 produces identical results in the floating point implementation.

Tangent_p1 = tan(x) + 1

```
//row5*(tangent+1)
r_ymm0 = _mm256_mul_ps(row5, tangent_p1);
//row3*(tangent+1)
r_ymm1 = _mm256_mul_ps(row3, tangent_p1);
```

This optimization could not be applied to the short integer implementation due to the scaling of numbers in the short algorithm.

Using Intel® Architecture Code Analyzer

The Intel® Architecture Code Analyzer is an excellent tool that identifies the critical path in a basic block, execution port utilization, and instruction usage alternatives. There is much more Intel® Architecture Code Analyzer information available at the Intel(R) AVX developer site.

The basic block analysis in Figure 11 shows that the vshufps (invoked via the *_mm256_shuffle_ps* intrinsic) executes on port 5. The analysis also shows that the vinsertf128 instruction (*_mm256_insertf128_ps* intrinsic) offers the programmer choices for port execution. The "X" in the Intel® Architecture Code Analyzer report indicates that an instruction can be executed on the port, but in this particular algorithm it was executed elsewhere. In this case, the vinsertf128 was executed on port 0 and port 3 rather than port 3 and port 5 because the second source operand was loaded from memory. The insert takes place on port 0, thus reducing the pressure on port 5.

Num of	Ports pressure in cycles									
Uops	0 - DV	1	2 - D	3 - D	4	5				
1				X	X	1	1			CP	vmovaps xmm11, xmmword ptr [r10+r8*4+0x20]
1				X	X	1	1			CP	vmovaps xmm4, xmmword ptr [r10+r8*4+0x30]
2	1			X	X	1	1		X	CP	vinsertf128 ymm14, ymm11, xmmword ptr [r10+r8*4+0xe0], 0x1
2	1			X	X	1	1		X	CP	vinsertf128 ymm13, ymm4, xmmword ptr [r10+r8*4+0xf0], 0x1
1									1	CP	vshufps ymm5, ymm14, ymm14, 0x0
2^	1			1	2	X	X			CP	vmulps ymm10, ymm5, ymmword ptr [rip+0x43482]
1									1	CP	vshufps ymm8, ymm14, ymm14, 0xaa
2^	1			1	2	X	X			CP	vmulps ymm15, ymm8, ymmword ptr [rip+0x43494]
1									1	CP	vshufps ymm9, ymm14, ymm14, 0x55
2^	1			X	X	1	2			CP	vmulps ymm11, ymm9, ymmword ptr [rip+0x434a6]
1			1							CP	vaddps ymm15, ymm10, ymm15

Figure 11 - Basic Block Analysis

The Intel® Architecture Code Analyzer is an effective optimization tool for software developers who want to fine tune and improve their application's performance.

Results

The 128-bit code was compiled for the Intel® microarchitecture codenamed Nehalem to generate Intel® SSE code, and executed on the Sandy Bridge microarchitecture-based silicon. The corresponding 256-bit Intel® AVX-enabled code was compiled for the Sandy Bridge microarchitecture and executed on the Sandy Bridge microarchitecture-based silicon. Data was aligned on 16-Byte boundaries for the Intel® SSE code and 32-Byte boundaries for the Intel® AVX code. Both applications were compiled using the 64-bit version of the Intel® C++ Compiler Professional Edition, version 11.1.038. The speedups listed here are for 256-bit code relative to the 128-bit code.

Table 1 - Performance Results

Algorithm	Speedup	Parameters
Intel® AVX floating point implementation	0.94x	Compared to Intel® SSE short integer implementation
Intel® SSE short integer implementation compiled with /QxAVX	1.07x	Compared to Intel® SSE short integer implementation compiled with /QxSSE4.1
Intel® AVX floating point implementation	1.78x	Compared to Intel® SSE floating point implementation

Table 2 summarizes the overall mean error results for each implementation.

Table 2 - Overall Mean Error Results

L, H (Reference 5)	Intel® SSE Short	Intel® SSE Single Precision Floating Point	Intel® AVX Single Precision Floating Point
256, 255	3.44e-5	6.25e-6	6.25e-6
5, 5	2.58e-4	1.56e-6	1.56e-6
300, 300	4.69e-6	6.25e-6	6.25e-6
-255, 256	7.53e-4	3.13e-6	3.13e-6
-5, 5	0	0	0
-300, 300	0	0	0
Input = 0	0	0	0

Both the short integer and single precision floating point implementations meet the error requirements of IEEE 1180-1900 (Reference 5). The floating point implementations produce lower overall mean error in many of the tests; 5x, 165x, and 240x lower than the short integer implementation.

Conclusion

The results for the IDCT of 10,000 8x8 blocks shows that the Intel® AVX version outperformed the Intel® SSE single precision floating point implementation by 1.78x. The accuracy is excellent when compared to the reference IDCT.

The results also show the Intel® SSE short integer version compiled with the /QxAVX option outperformed the exact same code compiled with the /QxSSE4.1 option by 1.07x. The assembly language produced when compiling with /QxSSE4.1 has 22 register-to-register moves. The code produced with the /QxAVX switch did not have any register-to-register moves. The Intel® AVX non-destructive source instructions reduce

the need for register copies in this application. There can be benefits to using Intel AVX for integer-based algorithms today.

Although the Intel® AVX single precision floating point implementation is slightly slower than the Intel® SSE short version, the Intel® AVX single precision floating point version is more accurate. The Intel® SSE short version required several adjustments to improve the accuracy and minimize rounding errors. Those adjustments were not necessary in either floating point implementation, resulting in a cleaner and more accurate implementation.

Source Code for IDCT

The complete source code for the IDCT can be downloaded from here

Partial source code listing is provided in the following section.

```
void idctAVX(void)     {
__m128 r_xmm0, r_xmm2, r_xmm1;
__m256 r_ymm02, r_ymm46, r_ymm13, r_ymm57;
__m256 r_ymm0, r_ymm1, r_ymm2, r_ymm3, r_ymm4, r_ymm5, r_ymm6, r_ymm7;
__m256 row0, row1, row2, row3, row4, row5, row6, row7;
__m256 ymm_even, ymm_odd, ymm_sum, ymm_diff;
__m256 temp3, temp7;
__m256 tangent_1, tangent_2, tangent_3, cos_4;

tangent_1 = AVX_tg_1_16;
tangent_2 = AVX_tg_2_16;
tangent_3 = AVX_tg_3p1_16;
cos_4 = AVX_cos_4p1_16;

const float * pInput;
float * pOutput;
float * pFTab_i_04 = float_tab_i_04;
float * pFTab_i_26 = float_tab_i_26;
float * pFTab_i_17 = float_tab_i_17;
float * pFTab_i_35 = float_tab_i_35;
const int blockSize = 8*8;

//Transform all the blocks N times
//The iteration count is a command line option
const int maxLoopCount = g_loopCount;
long startTime = getTimestamp();

for(int loopCount = 0; loopCount < maxLoopCount; loopCount++)     {

   //Operate on all the blocks
   for(int i = 0; i < g_blockCount; i++)  {

      //Get pointers for this input and output
      pInput = &dctData[i*blockSize];
      pOutput = &kernelResults[i*blockSize];
      pFTab_i_04 = float_tab_i_04;
      pFTab_i_26 = float_tab_i_26;
      pFTab_i_17 = float_tab_i_17;
      pFTab_i_35 = float_tab_i_35;
//    IACA_START
```

```
//Rows 0 and 4
//Process the first four floats of these two rows
//Read input data from row 0, read eight floats via two 128-bit loads
r_ymm0 = _mm256_castps128_ps256(_mm_load_ps(pInput));
r_ymm1 = _mm256_castps128_ps256(_mm_load_ps(&pInput[4]));

//Insert data from row 4 into the upper lane
r_ymm0 = _mm256_insertf128_ps(r_ymm0, _mm_load_ps(&pInput[4*8]), 1);
r_ymm4 = _mm256_insertf128_ps(r_ymm1, _mm_load_ps(&pInput[4*8+4]), 1);

//Broadcast float 0
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0x00);

//Multiply by the coefficients
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) pFTab_i_04));

//Broadcast float 2 and multiply by coefficients
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0xaa);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_04[8]));

//Add
r_ymm02 = _mm256_add_ps(r_ymm2, r_ymm3);

//Broadcast float 1 and multiply by coefficients
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0x55);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_04[16]));

//Broadcast float 3 and multiply by coefficients
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0xff);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_04[24]));

//Add
r_ymm13 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process the second four floats of these two rows
//Broadcast float 4
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0x00);

//Multiply by the coefficients
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_04[32]));

//Broadcast float 6 and multiply by coefficients
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0xaa);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_04[40]));

//Add
r_ymm46 = _mm256_add_ps(r_ymm2, r_ymm3);

//Broadcast float 5 and multiply by coefficients
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0x55);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_04[48]));

//Broadcast float 7 and multiply by coefficients
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0xff);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_04[56]));
```

```
//Add to create this partial sum,
//then create final sums and differences.
r_ymm57 = _mm256_add_ps(r_ymm2, r_ymm3);
ymm_even = _mm256_add_ps(r_ymm02, r_ymm46);
ymm_odd = _mm256_add_ps(r_ymm13, r_ymm57);
ymm_diff = _mm256_sub_ps(ymm_even, ymm_odd);
ymm_sum = _mm256_add_ps(ymm_even, ymm_odd);

//Reverse the order of the differences, then build outputs 0 and 4
ymm_diff = _mm256_shuffle_ps(ymm_diff, ymm_diff, 0x1b);
row0 = _mm256_permute2f128_ps(ymm_sum, ymm_diff, 0x20);
row4 = _mm256_permute2f128_ps(ymm_sum, ymm_diff, 0x31);
//End of Rows 0 and 4 processing

//Rows 1 and 7
//Process the first four floats of these two rows
//Read input data from row 1, read eight floats via two 128-bit loads
r_ymm0 = _mm256_castps128_ps256(_mm_load_ps(&pInput[8]));
r_ymm1 = _mm256_castps128_ps256(_mm_load_ps(&pInput[8+4]));

//Insert data from row 7 into the upper lane
r_ymm0 = _mm256_insertf128_ps(r_ymm0, _mm_load_ps(&pInput[7*8]), 1);
r_ymm4 = _mm256_insertf128_ps(r_ymm1, _mm_load_ps(&pInput[7*8+4]), 1);

//Process floats 0 and 2
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0x00);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) pFTab_i_17));
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0xaa);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_17[8]));
r_ymm02 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process floats 1 and 3
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0x55);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_17[16]));
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0xff);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_17[24]));
r_ymm13 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process the second four floats of these two rows
//Process floats 4 and 6
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0x00);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_17[32]));
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0xaa);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_17[40]));
r_ymm46 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process floats 5 and 7
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0x55);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_17[48]));
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0xff);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_17[56]));
r_ymm57 = _mm256_add_ps(r_ymm2, r_ymm3);

//Create final sums and differences
ymm_even = _mm256_add_ps(r_ymm02, r_ymm46);
ymm_odd = _mm256_add_ps(r_ymm13, r_ymm57);
ymm_diff = _mm256_sub_ps(ymm_even, ymm_odd);
```

```
ymm_sum = _mm256_add_ps(ymm_even, ymm_odd);

//Reverse the order of the differences, then build outputs 1 and 7
ymm_diff = _mm256_shuffle_ps(ymm_diff, ymm_diff, 0x1b);
row1 = _mm256_permute2f128_ps(ymm_sum, ymm_diff, 0x20);
row7 = _mm256_permute2f128_ps(ymm_sum, ymm_diff, 0x31);
//End of Rows 1 and 7 processing

//Rows 2 and 6
//Process the first four floats of these two rows
//Read input data from row 2, read eight floats via two 128-bit loads
r_ymm0 = _mm256_castps128_ps256(_mm_load_ps(&pInput[2*8]));
r_ymm1 = _mm256_castps128_ps256(_mm_load_ps(&pInput[2*8+4]));

//Insert data from row 6 into the upper lane
r_ymm0 = _mm256_insertf128_ps(r_ymm0, _mm_load_ps(&pInput[6*8]), 1);
r_ymm4 = _mm256_insertf128_ps(r_ymm1, _mm_load_ps(&pInput[6*8+4]), 1);
//Process floats 0 and 2
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0x00);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) pFTab_i_26));
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0xaa);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_26[8]));
r_ymm02 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process floats 1 and 3
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0x55);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_26[16]));
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0xff);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_26[24]));
r_ymm13 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process the second four floats of these two rows
//Process floats 4 and 6
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0x00);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_26[32]));
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0xaa);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_26[40]));
r_ymm46 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process floats 5 and 7
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0x55);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_26[48]));
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0xff);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_26[56]));
r_ymm57 = _mm256_add_ps(r_ymm2, r_ymm3);

//Create the final sum and difference
ymm_even = _mm256_add_ps(r_ymm02, r_ymm46);
ymm_odd = _mm256_add_ps(r_ymm13, r_ymm57);
ymm_diff = _mm256_sub_ps(ymm_even, ymm_odd);
ymm_sum = _mm256_add_ps(ymm_even, ymm_odd);

//Reverse the order of the differences, then build outputs 2 and 6
ymm_diff = _mm256_shuffle_ps(ymm_diff, ymm_diff, 0x1b);
row2 = _mm256_permute2f128_ps(ymm_sum, ymm_diff, 0x20);
row6 = _mm256_permute2f128_ps(ymm_sum, ymm_diff, 0x31);
//End of Rows 2 and 6 processing
```

```
//Rows 3 and 5
//Process the first four floats of these two rows
//Read input data from row 3, read eight floats via two 128-bit loads
r_ymm0 = _mm256_castps128_ps256(_mm_load_ps(&pInput[3*8]));
r_ymm1 = _mm256_castps128_ps256(_mm_load_ps(&pInput[3*8+4]));

//Insert data from row 5 into the upper lane
r_ymm0 = _mm256_insertf128_ps(r_ymm0, _mm_load_ps(&pInput[5*8]), 1);
r_ymm4 = _mm256_insertf128_ps(r_ymm1, _mm_load_ps(&pInput[5*8+4]), 1);

//Process floats 0 and 2
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0x00);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) pFTab_i_35));
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0xaa);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_35[8]));
r_ymm02 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process floats 1 and 3
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0x55);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_35[16]));
r_ymm1 = _mm256_shuffle_ps(r_ymm0, r_ymm0, 0xff);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_35[24]));
r_ymm13 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process the second four floats of these two rows
//Process floats 4 and 6
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0x00);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_35[32]));
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0xaa);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_35[40]));
r_ymm46 = _mm256_add_ps(r_ymm2, r_ymm3);

//Process floats 5 and 7
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0x55);
r_ymm2 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_35[48]));
r_ymm1 = _mm256_shuffle_ps(r_ymm4, r_ymm4, 0xff);
r_ymm3 = _mm256_mul_ps(r_ymm1, *((__m256 *) &pFTab_i_35[56]));
r_ymm57 = _mm256_add_ps(r_ymm2, r_ymm3);

//Create the final sum and difference
ymm_even = _mm256_add_ps(r_ymm02, r_ymm46);
ymm_odd = _mm256_add_ps(r_ymm13, r_ymm57);
ymm_diff = _mm256_sub_ps(ymm_even, ymm_odd);
ymm_sum = _mm256_add_ps(ymm_even, ymm_odd);

//Reverse the order of the differences, then build outputs 3 and 5
ymm_diff = _mm256_shuffle_ps(ymm_diff, ymm_diff, 0x1b);
row3 = _mm256_permute2f128_ps(ymm_sum, ymm_diff, 0x20);
row5 = _mm256_permute2f128_ps(ymm_sum, ymm_diff, 0x31);
//End of Rows 3 and 5 processing

//*****************************
//perform 1D IDCT on the columns

//Multiply several rows by the appropriate tangent value
//row5*(tangent3+1)
```

```
r_ymm0 = _mm256_mul_ps(row5, tangent_3);
//row3*(tangent3+1)
r_ymm1 = _mm256_mul_ps(row3, tangent_3);
//row7*tangent1
r_ymm4 = _mm256_mul_ps(row7, tangent_1);
//row1*tangent1
r_ymm5 = _mm256_mul_ps(row1, tangent_1);

//Begin to create results
//[row5*tangent3 + row5] + row3
r_ymm0 = _mm256_add_ps(r_ymm0, row3);
//row5 - [row1*tangent3 + row1]
r_ymm2 = _mm256_sub_ps(row5, r_ymm1);
//row6*tangent2
r_ymm7 = _mm256_mul_ps(row6, tangent_2);
//row2*tangent2
r_ymm3 = _mm256_mul_ps(row2, tangent_2);
//row1*tangent1 - row7
r_ymm5 = _mm256_sub_ps(r_ymm5, row7);
//row7*tangent1 + row1
r_ymm4 = _mm256_add_ps(r_ymm4, row1);

//Save intermediate row 7 results, used as an input later
//[row7*tangent1 + row1] + [row5*tangent3 + row3]
temp7 = _mm256_add_ps(r_ymm4, r_ymm0);

//Save intermediate row 3 results, used as an input later
//[row1*tangent1 - row7] + [row5 - [row1*tangent3 + row1]]
temp3 = _mm256_add_ps(r_ymm5, r_ymm2);

//[row7*tangent1 + row1] - [row5*tangent3 + row3]
r_ymm4 = _mm256_sub_ps(r_ymm4, r_ymm0);
//[row1*tangent1 - row7] - [row5 - [row1*tangent3 + row1]]
r_ymm5 = _mm256_sub_ps(r_ymm5, r_ymm2);
//{[row7*tangent1 + row1] - [row5*tangent3 + row3]} -
//{[row1*tangent1 - row7] - [row5 - [row1*tangent3 + row1]]}
r_ymm1 = _mm256_sub_ps(r_ymm4, r_ymm5);
//{[row7*tangent1 + row1] - [row5*tangent3 + row3]} +
//{[row1*tangent1 - row7] - [row5 - [row1*tangent3 + row1]]}
r_ymm4 = _mm256_add_ps(r_ymm4, r_ymm5);
//multiply by cos_4+1
r_ymm4 = _mm256_mul_ps(r_ymm4, cos_4);
//row6*tangent2 + row2
r_ymm7 = _mm256_add_ps(r_ymm7, row2);
//row2*tangent2 - row6
r_ymm3 = _mm256_sub_ps(r_ymm3, row6);
//multiply by cos_4+1
r_ymm0 = _mm256_mul_ps(r_ymm1, cos_4);
//row0 + row4
r_ymm5 = _mm256_add_ps(row0, row4);
//row0 - row4
r_ymm6 = _mm256_sub_ps(row0, row4);
//[row0 + row4] - [row6*tangent2 + row2]
r_ymm2 = _mm256_sub_ps(r_ymm5, r_ymm7);
//[row0 + row4] + [row6*tangent2 + row2]
r_ymm5 = _mm256_add_ps(r_ymm5, r_ymm7);
//[row0 - row4] - [row2*tangent2 - row6]
```

```
        r_ymm1 = _mm256_sub_ps(r_ymm6, r_ymm3);
        //[row0 - row4] + [row2*tangent2 - row6]
        r_ymm6 = _mm256_add_ps(r_ymm6, r_ymm3);
        //[[row7*tangent1 + row1] + [row5*tangent3 + row3]] +
        //[[row0 + row4] + [row6*tangent2 + row2]]
        r_ymm7 = _mm256_add_ps(temp7, r_ymm5);

        //Store row 0 results (store 1 of 8)
        _mm256_store_ps(pOutput, r_ymm7);
        //[[row0 - row4] + [row2*tangent2 - row6]] -
        //cos4*{[row7*tangent1 + row1] - [row5*tangent3 + row3]} +
        //{[row1*tangent1 - row7] - [row5 - [row1*tangent3 + row1]]}
        r_ymm3 = _mm256_sub_ps(r_ymm6, r_ymm4);
        //[[row0 - row4] + [row2*tangent2 - row6]] +
        //cos4*{[row7*tangent1 + row1] - [row5*tangent3 + row3]} +
        //{[row1*tangent1 - row7] - [row5 - [row1*tangent3 + row1]]}
        r_ymm6 = _mm256_add_ps(r_ymm6, r_ymm4);

        //Store row 1 results (store 2 of 8)
        _mm256_store_ps(&pOutput[1*8], r_ymm6);

        r_ymm7 = _mm256_sub_ps(r_ymm1, r_ymm0);
        r_ymm1 = _mm256_add_ps(r_ymm1, r_ymm0);

        r_ymm6 = _mm256_add_ps(r_ymm2, temp3);
        r_ymm2 = _mm256_sub_ps(r_ymm2, temp3);

        r_ymm5 = _mm256_sub_ps(r_ymm5, temp7);

        //Store final results
        _mm256_store_ps(&pOutput[2*8], r_ymm1);
        _mm256_store_ps(&pOutput[3*8], r_ymm6);
        _mm256_store_ps(&pOutput[4*8], r_ymm2);
        _mm256_store_ps(&pOutput[5*8], r_ymm7);
        _mm256_store_ps(&pOutput[6*8], r_ymm3);
        _mm256_store_ps(&pOutput[7*8], r_ymm5);

//      IACA_END
        }
    }
    long duration = getTimestamp() - startTime;
    cout << "AVX Timestamp = " << duration << endl;
}
```

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. Pennebaker and Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993, pp. 29-64.

2. *A Fast Precise Implementation of 8x8 Discrete Cosine Transform Using the Streaming SIMD Extensions and MMX™ Instructions*, Intel Application Note, AP-922, Copyright 1999
3. *Using Streaming SIMD Extensions 2 (SSE2) to Implement and Inverse Discrete Cosine Transform*, Intel Application Note, AP-945, Copyright 2000
4. Rao and Yip, *Discrete Cosine Transform Algorithms, Advantages, Applications*, Academic Press, Inc., Boston, 1990, Appendix A.2
5. IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform, IEEE Std 1180-1990.

About the Author

Richard Hubbard is a Senior Software Engineer and member of the SSG Apple enabling team, working on optimizing Mac OS X* applications for power and performance. Richard holds a Masters degree in Electrical Engineering from Stevens Institute of Technology and a Bachelors in Computer Engineering from New Jersey Institute of Technology.

perform, copy, make, have made, use, sell, and otherwise dispose of Intel's and its sub-licensee's products embodying such comments and suggestions in any manner and via any media Intel chooses, without reference to the source.

Copies of documents, which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel 's Web Site at www.intel.com.