1. Introduction

The AltiVec extensions to the PowerPC architecture include single-instruction, multiple-data (SIMD) instructions. This note presents a code example that exploits these instructions using the AltiVec C/C++ programming model to apply an inverse discrete cosine transformation to values in the context of ITU-T Recommendation H.263. It should be noted that the accuracy of this algorithm does not strictly meet the H.263 Annex A specification [1].

The inverse discrete cosine transform (IDCT) decodes an image into the spatial domain from a representation of the data better suited to compaction. IDCT-based decoding forms the basis for current image and video decompression standards. In H.263, the input to the IDCT comes after the dequantization step and zig-zag positioning. An 8x8 block of input values range from -2048 to 2047 and output values in the range -256 to 255. This information is used to reconstruct the image. An original image has no prediction applied and is labeled as an I-picture (INTRA) in the standard. Its pixel values range from 0 to 255. A difference image (INTER) has prediction applied and is labeled either as a P-picture or B-picture (when bi-directional prediction occurs). Its pixel values range from -255 to 255. The actual formula used depends on whether QP is even or odd, which is specified by the standard to prevent the accumulation of IDCT mismatch errors.

The 2D inverse discrete cosine transform is given by the following formula:

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

where:

- \(x, y\) = spatial coordinates in the pixel domain (0,1,2...7)
- \(u, v\) = coordinates in the transform domain (0,1,2...7)
- \(C_u = \frac{1}{\sqrt{2}}\) for \(u=0\), otherwise 1
- \(C_v = \frac{1}{\sqrt{2}}\) for \(v=0\), otherwise 1

The separable nature of the 2D IDCT is exploited by performing a 1D IDCT on the eight columns and then a 1D IDCT on the eight rows of the result.
Several fast algorithms are available to calculate the 8 point 1D IDCT. The algorithm implemented in AltiVec is a scaled version of Chen\[2\] developed by the IBM's Haifa Research Laboratory [3] shown in Figures 1 and 2. It was selected due to its minimization of the number of adds and multiplies required to arrive at the result.

To optimize Chen's algorithm for scalability and the AltiVec instruction set, the following identity was used:

\[ ax + by = a(x + (b/a)y) \]

---

**Figure 1 - Stages of Modified Scaled Chen 1D IDCT**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>c4*c4/4</td>
<td>c4*c1/4</td>
<td>c4*c2/4</td>
<td>c4*c3/4</td>
<td>c4*c4/4</td>
<td>c4*c3/4</td>
<td>c4*c2/4</td>
<td>c4*c1/4</td>
</tr>
<tr>
<td>1</td>
<td>c1*c4/4</td>
<td>c1*c1/4</td>
<td>c1*c2/4</td>
<td>c1*c3/4</td>
<td>c1*c4/4</td>
<td>c1*c3/4</td>
<td>c1*c2/4</td>
<td>c1*c1/4</td>
</tr>
<tr>
<td>2</td>
<td>c2*c4/4</td>
<td>c2*c1/4</td>
<td>c2*c2/4</td>
<td>c2*c3/4</td>
<td>c2*c4/4</td>
<td>c2*c3/4</td>
<td>c2*c2/4</td>
<td>c2*c1/4</td>
</tr>
<tr>
<td>3</td>
<td>c3*c4/4</td>
<td>c3*c1/4</td>
<td>c3*c2/4</td>
<td>c3*c3/4</td>
<td>c3*c4/4</td>
<td>c3*c3/4</td>
<td>c3*c2/4</td>
<td>c3*c1/4</td>
</tr>
<tr>
<td>4</td>
<td>c4*c4/4</td>
<td>c4*c1/4</td>
<td>c4*c2/4</td>
<td>c4*c3/4</td>
<td>c4*c4/4</td>
<td>c4*c3/4</td>
<td>c4*c2/4</td>
<td>c4*c1/4</td>
</tr>
<tr>
<td>5</td>
<td>c5*c4/4</td>
<td>c5*c1/4</td>
<td>c5*c2/4</td>
<td>c5*c3/4</td>
<td>c5*c4/4</td>
<td>c5*c3/4</td>
<td>c5*c2/4</td>
<td>c5*c1/4</td>
</tr>
<tr>
<td>6</td>
<td>c6*c4/4</td>
<td>c6*c1/4</td>
<td>c6*c2/4</td>
<td>c6*c3/4</td>
<td>c6*c4/4</td>
<td>c6*c3/4</td>
<td>c6*c2/4</td>
<td>c6*c1/4</td>
</tr>
<tr>
<td>7</td>
<td>c7*c4/4</td>
<td>c7*c1/4</td>
<td>c7*c2/4</td>
<td>c7*c3/4</td>
<td>c7*c4/4</td>
<td>c7*c3/4</td>
<td>c7*c2/4</td>
<td>c7*c1/4</td>
</tr>
</tbody>
</table>

**Figure 2 - Post Scaling Matrix Using:** \[ cn = \cos\left(\frac{n\pi}{16}\right) \]
2. AltiVec Implementation

The implementation of the above algorithm requires the following steps:

- Transpose the matrix.
- Load the data and multiplication constants.
- Load the pre scaling matrix (Figure 2) and prescale the input.
- Perform the IDCT on the eight columns according to the stages shown in Figure 1.
- Transpose the matrix.
- Perform the IDCT on the eight rows according to the stages shown in Figure 1.

Because the previous step in H.263 revolves around zig-zag positioning, by reordering how the position is performed, the transposition is available at no additional cost.

Because there are several large constants to be setup, there are multiple ways this can be accomplished. The following two methods were considered:

1) Set up compile-time load of a vector of the different constants and then vec_splat them into the individual vectors.

```c
vector signed short SpecialConstants =

(vector signed short) (23170, 13753, 6518, 21895, -23170, -21895, 0, 0);

c4 = vec_splat(SpecialConstants, 0); /* c4 = cos(4*pi/16) */
a0 = vec_splat(SpecialConstants, 1); /* a0 = c6/c4 */
a1 = vec_splat(SpecialConstants, 2); /* a1 = c7/c1 */
...
```

2) Explicitly load each of the vector with the appropriate constant.

```c
c4 = (vector signed short)(23170); /* c4 = cos(4*pi/16) */
a0 = (vector signed short)(13753); /* a0 = c6/c4 */
a1 = (vector signed short)(6518); /* a1 = c7/c1 */
...
```

Choosing the methods for your application depends on the other vector operations being employed and determining which best distributes work to the various vector functional units. The first method distributes the work to the permutation unit after the original load. The second method requires a load for every constant and places more of a demand on the load/store unit.

Prescaling is done by using vec_mradds() to obtain the high order 16 bits of the 32 bit intermediate multiplication result using the output results and the PreScale matrix. The PreScale matrix is obtained by multiplying $2^{16}$ times the results of the equations contained in Figure 2.

The one dimensional IDCT is implemented in stages as labeled in Figure 1. The code that implements these is grouped to correspond to the various stages. For clarity, input values are labeled i0-i7, stage 1 values are labeled r0-r7, stage 2 values are labeled s0-s7, stage 3 values are labeled t0-t7, and output values are labeled o0-o7.
/* stage 1 */

r0 = i0;
r1 = i4;
r2 = i2;
r3 = i6;

temp = vec_mradds(a1, i1, (vector signed short)(0));
r4 = vec_subs(temp, i7);
r5 = vec_mradds(a2, i3, i5);
r6 = vec_mradds(a2, i5, i3);
r7 = vec_mradds(a1, i7, i1);

/* stage 2 */

s0 = vec_adds(r0, r1);
s1 = vec_subs(r0, r1);

s2 = vec_adds(s0, r2, (vector signed short)(0));
s3 = vec_adds(r0, r2);

s4 = vec_adds(r4, r5);
s5 = vec_adds(r4, r5);
s6 = vec_adds(r7, r6);
s7 = vec_adds(r7, r6);

/* stage 3 */

t0 = vec_adds(s0, s3);
t1 = vec_adds(s1, s2);
t2 = vec_adds(s1, s2);
t3 = vec_adds(s0, s3);

s4 = vec_adds(r4, r5);
s5 = vec_adds(r4, r5);
s6 = vec_adds(r7, r6);
s7 = vec_adds(r7, r6);

/* Output */

o0 = vec_adds(t0, t7);
o1 = vec_mradds(c4, t6, t1);
o2 = vec_mradds(c4, t5, t2);
o3 = vec_adds(t3, t4);
o4 = vec_adds(t3, t4);
o5 = vec_mradds(mcr, t6, t1);
o6 = vec_mradds(mcr, t6, t1);
o7 = vec_adds(t0, t7);
The above example can be optimized by removing direct assignments and reusing variables. All stages have been fully enumerated for clarity. It is shown in a more efficient form in the C source code. This code was made into a macro, however it could also have been made into an INLINE function, like the matrix transpose. Using macros minimizes the penalty for the call overhead, subject to compiler limitations.

The matrix transpose uses `vec_mergeh()` and `vec_mergel()` for the transposition. C source is "2DDCTMTCSCOD" on web page under Design Tools/Software.

3. Code Samples

C source for the IDCT function is "2IDCTCCOD " on web page under Design Tools/Software.

Assembly output of the compiler is "2IDCTSCOD" on web page under Design Tools/Software.

4. Performance

Performance results are given in clock cycles for a typical AltiVec implementation. Performance for a specific AltiVec implementation may vary. Also, performance can vary depending on the C compiler used and can improve as the quality of a compiler improves from release to release. It is assumed that all required instructions and data are present in the L1 cache.

The call to the function IDCT takes 103 cycles to calculate a 2D IDCT on an 8x8 block of coefficients. This time includes the function call overhead.

5. References

