1. Introduction

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

The discrete cosine transform (DCT) encodes an image from the spatial domain into a representation of the data better suited to compaction. DCT-based encoding forms the basis for current image and video compression standards. In H.263, the input to the DCT comes from motion compensation (as a difference image) or the original video stream depending on the coding control specification. 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 input data is an 8x8 block of half words, and the output is an 8x8 block of halfwords with values ranging from -2040 to 2040.

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

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

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 DCT is exploited by performing a 1D DCT on the eight columns, and then a 1D DCT on the eight rows of the result.
Several fast algorithms are available to calculate the 8-point 1D DCT. 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 & 2. It was selected due to its minimizing 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 DCT**

<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:** $c_n = \cos \left( \frac{2\pi n}{16} \right)$
2. AltiVec Implementation

The implementation of the above algorithm requires the following steps:

- Load the data and multiplication constants.
- Perform the DCT on the eight columns according to the stages shown in Figure 1.
- Transpose the matrix.
- Perform the DCT on the eight rows according to the stages shown in Figure 1.
- Load the post scaling matrix, post-scale the output by multiplying it by the matrix, and store the result.

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

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

\[
\text{vector signed short SpecialConstants} = (\text{vector signed short}) (23170, 13753, 6518, 21895, -23170, -21895, 0,0);
\]

\[
c4 = \text{vec_splat}(\text{SpecialConstants}, 0); /* c4 = \cos(4\pi/16) */
\]

\[
a0 = \text{vec_splat}(\text{SpecialConstants}, 1); /* a0 = c6/c4 */
\]

\[
a1 = \text{vec_splat}(\text{SpecialConstants}, 2); /* a1 = c7/c1 */
\]

... Explicitly load each vector with the appropriate constant.

\[
c4 = (\text{vector signed short})(23170); /* c4 = \cos(4\pi/16) */
\]

\[
a0 = (\text{vector signed short})(13753); /* a0 = c6/c4 */
\]

\[
a1 = (\text{vector signed short})(6518); /* a1 = c7/c1 */
\]

... Choosing between these methods for your application depends on the other vector operations being used 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.

The one dimensional DCT 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 = vec_adds(i0, i7);
r7 = vec_subs(i0, i7);
r1 = vec_adds(i1, i6);
r6 = vec_subs(i1, i6);
r2 = vec_adds(i2, i5);
r5 = vec_subs(i2, i5);
r3 = vec_adds(i3, i4);
r4 = vec_subs(i3, i4);

/* Stage 2 */
s0 = vec_adds(r0, r3);
s3 = vec_subs(r0, r3);
s1 = vec_adds(r1, r2);
s2 = vec_subs(r1, r2);
s4 = r4;
s7 = r7;
s5 = vec_subs(t6, t5);
s6 = vec_adds(t6, t5);

/* Stage 3 */
t0 = vec_adds(s0, s1);
t1 = vec_subs(s0, s1);
t2 = vec_mradds(s2, a0, s3);
temp = vec_mradds(s3, a0, (vector signed short)(0));
t3 = vec_subs(temp, s2);
t4 = vec_mradds(s5, c4, s4);
t5 = vec_mradds(s5, mc4, s4);
t6 = vec_mradds(s6, mc4, s7);
t7 = vec_mradds(s6, c4, s7);

/* Stage 4 */
o0 = t0;
o4 = t1;
o2 = t2;
o6 = t3;
o1 = vec_mradds(t4, a1, t7);
temp = vec_mradds(t7, a1, (vector signed short)(0));
o7 = vec_subs(temp, t4);
o5 = vec_mradds(t6, a2, t5);
o3 = vec_mradds(t5, ma2, t6);
The above example can be optimized by removing direct assignments and reusing variables. It has all stages fully enumerated for clarity. It is shown in a more efficient form in the C source code. This code was made into a macro in the source, however it could also have been made into an INLINE function, like the matrix transpose. Using these methods minimizes the penalty for the call overhead, subject to compiler limitations.

The matrix transpose relies on using `vec_mergeh()` and `vec_mergel()` calls to effect the transpose. C source for this function is "2DDCTMTCSCOD" on the web.

Postscaling 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 PostScale matrix. The PostScale matrix is obtained by multiplying with the results of the equations contained in Figure 2.

3. Code Samples

C source for the function ("2DDCTTFCCOD" on the web)

Assembly output of the compiler ("2DDCTMTSCOD" on the web)

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 in the L1 cache.

The call to the function DCT takes 102 cycles to calculate a 2D DCT on an 8x8 block of coefficients. This time includes the function call overhead.

5. References

