IMPLEMENTATION OF H.264 DECODER ON SANDBLASTER DSP

Vaidyanathan Ramadurai, Sanjay Jinturkar, Mayan Moudgill, John Glossner

Sandbridge Technologies, 1 North Lexington Avenue, White Plains, NY 10601
sjinturkar@sandbridgetech.com

ABSTRACT

This paper presents the optimization techniques and results of implementing the H.264/AVC baseline profile decoder in software on the Sandblaster digital signal processor. It has been implemented in ANSI C and optimized to exploit the architectural features of the processor. The software implementation enables the reusability of the processor and lowers the development costs.

1. INTRODUCTION

The motion pictures video group has developed a new video coding standard called H.264/AVC/ MPEG-4 Part 10 that provides better compression than the earlier standards such as MPEG4 and H.263 [7].

The H.264 concentrates on compression and transmission efficiency, and focuses on popular applications of video compression such as video telephony, digital TV etc. It provides three different profiles, namely the baseline profile (used for video conferencing), the main profile (used for video streaming across networks) and the extended profile (for video broadcast and storage) [5].

In this paper, we focus primarily on software implementation and optimization of the baseline profile in ANSI C. These optimizations can be extended to other profiles also.

In Section 2, we discuss the Sandblaster multithreaded processor, and Section 3 onwards we discuss the optimizations for baseline profile.

2. SANDBLASTER DSP

Sandbridge Technologies has developed the Sandblaster architecture for a convergence device. As handsets are converging to multimedia multi-protocol systems, the Sandblaster architecture supports the data types necessary for convergence devices including RISC control code, DSP, and Java.

As shown in Figure 1, the design includes a unique combination of modern techniques such as a SIMD Vector/DSP unit, a parallel reduction unit, and a RISC-based integer unit. Each processor core provides support for concurrent execution for up to eight threads of execution. All states may be saved from each individual thread and no special software support is required for interrupt processing. The machine is partitioned into a RISC-based control unit that fetches instructions from a set-associative instruction cache. Instruction space is conserved through the use of compounded instructions that are grouped into packets for execution.

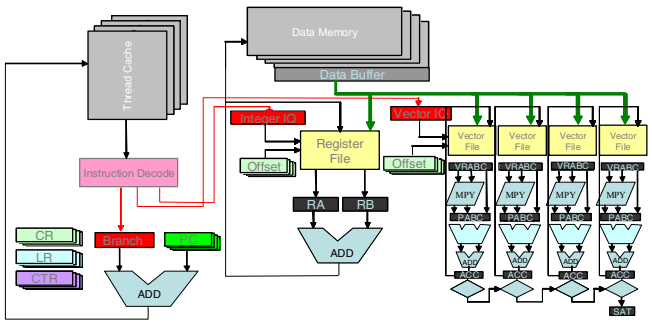

Figure 1: Sandblaster Multithreaded Processor

The memory subsystem has been designed carefully to minimize power dissipation. The pipeline design in combination with the memory design ensures that all memories are single ported and yet the processor can sustain nearly 4 taps per cycle for a filter (the theoretical maximum) in every thread unit simultaneously. A RISC-based execution unit, depicted in the center of Figure 1, assists with control processing.

Video codec processing often consists of control structures with compute-intensive inner loops. For the control code, a 16 entry, 32-bit register file per thread unit provides for very efficient control processing. Common integer data types are typically stored in the register file. This allows for branch bounds to be computed and addresses to be efficiently generated.

0-7803-9332-5/05/$20.00 ©2005 IEEE

Intensive loop processing is performed in the SIMD/Vector unit depicted on the right side of Figure 1. Each cycle, a 4x16-bit vector may be loaded into the register file while two vectors are being multiplied, saturated, reduced (e.g. summed), and saturated again. The branch bound may also be computed and the instruction looped on itself until the entire vector is processed. This may be specified in as little as 64-bits.

To enable multimedia processing in software, the processor supports many levels of parallelism. Thread-level parallelism is supported by providing hardware support for up to 8 independent programs to be simultaneously active on a single Sandblaster core. This minimizes the latency in physical layer processing. Since many algorithms have stringent requirements on response time, multithreading is an integral technique in reducing latencies. The data-level parallelism (SIMD) is supported through the use of a Vector unit. In the inner kernel of video codec routines such as DCT and SAD, the computations appear as vector operations of moderate length. In addition, the compound word instruction set provides instruction level parallelism.

3. H.264 DECODER IMPLEMENTATION

An open source reference H.264 decoder [8] was used for development. The reference decoder was compiled, executed and profiled using the Sandblaster software tools [2]. Using the profile information, the compute intensive blocks were identified, and then algorithmic and coding changes were applied to these blocks to achieve the maximum performance. All the changes were applied strictly at the C level and no assembly language or intrinsics were used. The modified code was recompiled, executed and profiled using the tools again.

The software changes maximize the usage of vector instructions and parallel instructions, minimize 32x32 multiplications, minimize division, and pre-compute the results in look up tables when possible The modified software required ~80% less cycles than the original code and executes on a fraction of the processing power available on the Sandblaster DSP.

The key routines that contributed to the high execution cost in the original code were:

- Inverse 4x4 integer transform
- Interpolation to compute the half and quarter pixel resolution
- Generation of macro block address coordinates
- Deblocking filter, used to minimize the blocking distortion
- Intra prediction of the macro-blocks

3.1. 4 x 4 INTERGER TRANSFORM

The inverse integer transform is performed on 4x4 blocks of luminance and chrominance value differences. All operations are carried out using integer arithmetic without any loss of accuracy.

The inverse 4x4 block transform can be vectorized on the platform by changes to the algorithm. The change ensures that loads and stores are being done from consecutive locations in memory. The transform typically has a horizontal and a vertical step. The code below shows the original and optimized code for the horizontal step.

The optimized code is completely vectorized by the compiler. The original code requires 16 additions, 8 shifts and 32 memory loads. The optimized code requires 4 additions, 2 shifts and 4 memory loads.

Horizontal stage (original)

```
for (j0=0;j<BLOCK_SIZE;j++)
{
    for (i0=0;i<BLOCK_SIZE;i++)
    {
      m5[i]=img->cof[i0][j0][i][j];
    }
   m6[0]=(m5[0]+m5[2]);
    m6[1]=(m5[0]-m5[2]);
    m6[2]=(m5[1]>>1)-m5[3];
    m6[3]=m5[1]+(m5[3]>>1);
}
```

The original code cannot be vectorized. However, it can be rewritten with slight modifications as below:

```
void itrans_stage1_horiz(short m[], short
m5[]){
    m[0]=(m5[0]+m5[8]);
    m[4]=(m5[0]-m5[8]);
    m[8]=(m5[4]>>1)-m5[12];
    m[12]= m5[4]+(m5[12]>>1);
}

short t1[16];

for (j=0;j<BLOCK_SIZE;j++)
{
   t2[i]=img->cof[i0][j0][i][j];

}
for (j=0;j<BLOCK_SIZE;j++) {

   itrans_stage1_horiz(&t1[j], &t2[j]);

}
```

The compiler will unroll the second loop with index j , inline the itrans_stage1_horiz function and then vectorize the operations. The process for first four iterations is shown below:

```
   //2 vec ld, 1 vec st, 1 vec add
   m[0]=(m5[0]+m5[8]);
   m[1]=(m5[1]+m5[9]);
   m[2]=(m5[2]+m5[10]);
   m[3]=(m5[3]+m5[11]);
   //2 vec ld, 1 vec st, 1 vec sub
   m[4]=(m5[0]-m5[8]);
   m[5]=(m5[1]-m5[9]);
   m[6]=(m5[2]-m5[10]);
   m[7]=(m5[3]-m5[11]);
   //2 vec ld, 1 vec shr,1 vec sub,1 vec
st
   m[8]=(m5[4]>>1)-m5[12];
   m[9]=(m5[5]>>1)-m5[13];
   m[10]=(m5[6]>>1)-m5[14];
   m[11]=(m5[7]>>1)-m5[15];
//2 vec ld, 1 vec shr, 1 vec add, 1 vec st
   m[12]= m5[4]+(m5[12]>>1);
   m[13]= m5[5]+(m5[13]>>1);
   m[14]= m5[6]+(m5[14]>>1);
   m[15]= m5[7]+(m5[15]>>1);
}
```

This approach provides ~75% improvement over the original number of cycles.

3.2. INTERPOLATION

H.264 supports integer, half-pixel and quarter-pixel interpolation for motion compensation. The motion vector precision is one-quarter pixel for luma and one eighth pixel in chroma. The interpolation is used to determine the intensity values at non-integer pixel positions. The input values to the interpolation process either lies within the image boundary or outside the image boundary. In the latter case, the edge pixels are repeated. A six tap filter is used to interpolate the samples. The original code shows how the pixels are interpolated horizontally for half pixel positions. The images at quarter-pixel positions are then obtained by averaging the samples at integer and half-pixel positions.

Two inefficiencies with the interpolation code can be improved upon. The following is the original interpolation code.

```
for (j = 0; j < BLOCK_SIZE; j++) {
  for (i = 0; i < BLOCK_SIZE; i++) {
    for (result = 0, x = -2; x < 4; x++)
       result +=
       imageY[max(0,min(maxold_y,y_pos+j))]
       [max(0,min(maxold_x,x_pos+i+x))]*COE
       F[x+2];
    block[i][j] = max(0, min(255,
(result+16)/32));
  }
}
```

First, a considerable amount (i.e. conditional statements) of computation goes into calculating whether a required pixel is within the frame or image boundary. As a side-effect, the multiply accumulates involved in the filter do not vectorize as there are conditional statements inside the loop. To simplify the computation, the loop is split into two separate loops, one where all the inputs for filtering lie within the boundary of the image and one where the inputs lie beyond the image boundary. Now the first loop will be vectorized.

In the case where the pixels might not fit into an image, the edge pixels are used. In this case, we avoid the (min, max) computation by using a look-up table. The lookup table provides the index of the edge pixel when the input pixel for interpolation falls beyond the image boundary. Note that the dimensions of the image are known when the header is decoded (e.g. . QCIF = 176, CIF = 352).

Secondly, the filter can be better vectorized if the loop bound is a multiple of 4. Therefore, we increase the size of the filter from six tap to eight tap, with the values for the seventh and eighth tap being 0. This enables the filter loop to be vectorized efficiently.

The above two changes contribute about 80% improvement over the original code.

3.3. Macroblock position

A significant amount of division and modulo operations are used in calculating the x and y macroblock coordinates for a given macroblock address. As the macroblock width of the picture and the macroblock address range is fixed for a given image type (e.g. CIF: MB width = 22, MB address = {0… 395}, QCIF: MB width = 11, MB address = {0…98}), these values can be precomputed and stored in a table. The table will look as shown in the following figure

MB address	X coordinate	Y coordinate
0	0	0
1	1	0
2	2	0
….	…	…
96	8	8
97	9	8
98	10	8

This reduces the computation of the macroblock coordinates by over 95%, as it eliminates the division and modulus operations to compute the coordinates.

3.4. Deblocking filter

The deblocking filter is applied to all 4x4 block edges of a picture, except edges at the boundary of the picture and any edges for which the deblocking filter process is disabled. This filtering process is performed to eliminate blocking distortion on the picture edges. The filtering process is done in increasing order of the macroblock addresses. The deblocking of a macroblock requires macro blocks from above (if available) and left (if available).

The deblocking filter process is invoked for the luma and chroma components separately. For each macroblock, vertical edges are filtered first, from left to right, and then horizontal edges are filtered from top to bottom. The luma deblocking filter process is performed on four 16-sample edges and the chroma deblocking filter process is performed on two 8-sample edges. Sample values above and to the left of the current macroblock are used as input and may be further modified during the filtering of the current macroblock. Sample values modified during filtering of vertical edges are used as input for the filtering of the horizontal edges for the same macroblock.

The pixel strengths are calculated for 4 strips in the vertical and horizontal directions. The values of the strength could be in the range of 0-4. Even though there are a number of conditional statements that decide the filter strength for each pixel, all the 4 pixels in a given 4x4 block will always have the same strength. This is exploited to vectorize a significant portion of the code where the filter strengths are computed.

For example, if the block edge is not a macroblock edge and if the samples are in a macroblock coded using an Intra macroblock prediction mode, filter strength of 3 is used.

```
    if(edge!=      MB_EDGE      &&      (MbQ-
>mb_type==I4MB  ||MbQ->mb_type==I16MB||MbQ-
>mb_type==IPCM)) {
          for( idx=0 ; idx<16 ; idx++)
          {
              Strength[idx] = 3;
          }
  }
```

The above loop can be vectorized. The vectorization and associated optimizations provide a 75% improvement in cycles.

3.5. INTRA-PREDICTION

In intra prediction, only spatial redundancies are exploited to encode a video picture. These encoded frames are called I frames. Adjacent blocks in a frame tend to have similar properties and this is exploited to encode a frame. The difference between the actual block and the predicted block is directly transformed and encoded. The decoder, depending on the type of prediction adds the predicted samples with the inverse transformed differences to get the reconstructed block.

H.264 offers nine modes of intra prediction for 4x4 blocks of luminance data that includes a DC mode and 8 directional modes. The vertical prediction is discussed here. The DC and horizontal prediction can also be vectorized in a similar fashion.

VERTICAL PREDICTION

A	B	C	D	E	F	G	H	I
J	a	b	c	d				
K	e	f	g	h				
L	i	j	k	l				
M	m	n	o	p				

If vertical prediction is used, pixels {a, e, i, m} are equal to B, {b, f, j, n} are equal to C, {c, g, k, o} are equal to D and {d, h, l, p} are equal to E. The code can be written such that the copying of the pixel values is vectorized.

4. SUMMARY

In this paper, we have described the optimizations of the software implementation of the H.264 baseline profile in ANSI C . The optimizations have been done to exploit the architectural features of the processor.

5. REFERENCES

[1] Sandbridge Technologies, “Sandblaster DSP Overview”, www.sandbridgetech.com

[2] Sanjay Jinturkar , John Glossner, Mayan Moudgill, Erdem Hokenek, “Programming the Sandblaster Multithreaded Processor”, GSPx 2003.

[3] Sanjay Jinturkar , John Glossner, Vladimir Kotlyar, Erdem Hokenek, “The Sandblaster Automatic Multithreaded Vectorizing Compiler”, GSPx 2004.

[4] www.netbeans.org, “Netbeans IDE”.

[5] Iain E.G. Richardson, “H.264 and MPEG-4 video compression”, Wiley 2003.

[6] Juyup Lee, Sungkun Moon and Wonyong Sung, "H.264 Decoder Optimization Exploiting SIMD Instructions," IEEE Asia-Pacific Conference on Circuits and Systems, (APCCAS), December 2004.

[7] H.264 standard, DRAFT ISO/IEC 14496-10.

[8] http://iphome.hhi.de/suehring/tml/,“H.264/AVC Reference Software”.