Techniques for Optimization of Audio Codecs on NEON

This paper discusses the various optimization techniques for implementation of Audio Codecs on ARM Cortex-A8 processor solutions using the NEON technology.

By Karthick J, Ittiam Systems

The ARM Cortex™-A8 processor is the latest high-performance, power-efficient processor available from ARM. Based on the ARMv7 architecture, it is ARM's first superscalar processor featuring technology for enhanced code density and performance. An important feature of the Cortex-A8 processor is the NEON™ technology, which makes it suitable for multimedia processing. The Cortex-A8 processor generally operates at speeds ranging from 600 MHz to 1 GHz and can support SD and HD Video. A HD Video application running on the Cortex-A8 processor leaves little space for audio processing which requires the audio codecs to be optimized to use least resources.

NEON in the Cortex-A8 processor is designed to suit video/image signal processing algorithms and operates on vectorized data. Here, a vector refers to a data array with array length varying from 2 to 16. Any single operation such as an addition or a load/store is performed over a vector data in NEON.

In an audio codec most of the processing is done in a loop over scalar data array of length varying from 64 to 4096. The key challenge in codec optimization is to utilize the vector processing efficiently.

Processing on NEON is more efficient when the datatype of the operands are either 'char'(8 bit) or 'short'(16 bit) rather than 'int'(32 bit). For example, multiplication of a four-element 'short' array, takes 1 cycle, whereas a multiplication of a two-element 'int'(32 bit) array takes 2 cycles. In audio codecs data is needed to be maintained in 32 bits for a good audio quality. This makes efficient audio processing more challenging on NEON. This paper discusses some of the techniques to get an efficient implementation of Audio codecs on Cortex-A8 using NEON. Though the paper discusses processing blocks of audio codec as an example, the optimization techniques are generic to be used in any multimedia codec.

Optimization Techniques

The optimization techniques are discussed here are from the point of view of a decoder. However, they are applicable for an encoder as well. The generic block diagram of an audio decoder is shown in the figure below:

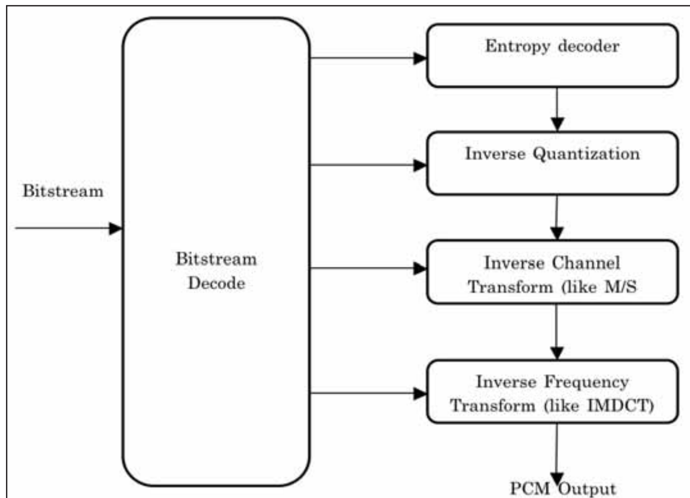

Figure 1: Functional Block Diagram of a typical Audio Decoder

Decoding of the quantized spectral coefficients and the quantization step sizes (usually called as scale factor or weight factor), are done by the Entropy decoder. After Entropy decoding inverse quantization is performed. In audio encoding a linear transform is applied on a multichannel data (2 or more channels), to utilize inter-channel redundancies. The corresponding inverse transform is applied in the decoding side. Finally, the spectral domain data is converted to time domain. The inverse frequency transforms is performed at the decoder using frequency transforms like Modified Discrete Cosine Transform (MDCT), Pseudo Quadrature Mirror Filters (PQMF).

Basic Technique for Vectorization

As NEON operates only on a vector, there needs to be same operation done multiple times so that they can be processed in a vector. The basic technique for vectorization of any given loop is that "**one should vectorize the code so that multiple iterations in a loop can occur in parallel rather than trying to merge the operations done in a single iteration**".

Consider the loop given below:

```
for i = 0 to n,
{
        a[i] = b[i] + c[i];
        d[i] = a[i] + e[i]
}
```

This loop could be vectorized as shown below:

```
for i = 0 to n/4,
{
/* The below four additions can be done using one vector
instruction in NEON */
        a[4*i] = b[4*i] + c[4*i];
        a[4*i + 1] = b[4*i + 1] + c[4*i + 1];
        a[4*i + 2] = b[4*i + 2] + c[4*i + 2];
        a[4*i + 3] = b[4*i + 3] + c[4*i + 3];

/* The below four additions can be done using one vector
instruction in NEON */
        d[4*i] = a[4*i] + e[4*i]
        d[4*i + 1] = a[4*i + 1] + e[4*i + 1]
        d[4*i + 2] = a[4*i + 2] + e[4*i + 2]
        d[4*i + 3] = a[4*i + 3] + e[4*i + 3]
}
```

Vectorization across multiple iterations is possible only when the operations performed in a particular iteration are independent of the results of the previous iteration. This means a function/module should be chosen for NEON implementation only if it is algorithmically possible to run multiple iterations in parallel. In an audio decoder the Entropy decoder block cannot be vectorized across multiple iterations because the decoding of second symbol can be started only after the completing of decode of first symbol. Most of the other blocks in audio processing can be modified to operate on a vector data.

High Precision Multiplications

Multiplications consume a significant amount of processor resource in audio codecs. In 'Inverse Quantization' the data coefficients needs to be multiplied by the weight factors (scale factors). IMDCT and PQMF involve lot of multiplication of data coefficients with sin/cos and window coefficients.

In NEON processor a multiplication of a four-element 'short' (16-bit) array, takes 1 cycle, whereas a multiplication of a two-element 'int'(32-bit) array takes 2 cycles. This makes multiplication of two 32-bit values costlier in NEON. Keeping the data values in 16-bits will severely affect the audio quality. Though it is necessary to maintain the width of data coefficients at 32-bits, the width of the other operand can be maintained in 16-bits without affecting the audio quality. In other words, the width of weight factors, sin/cos coefficients and window coefficients can be maintained in 16 bits.

So now, we need to multiply a 32-bit value with a 16-bit value (32x16 multiplication). NEON supports instructions only for multiplying two 16-bit values (16x16 multiplication) or two 32-bit values (32x32 multiplication). There is no direct way of performing 32x16 multiplications in NEON. But a 32x16 multiplication can be performed via 16x16 multiplications. In NEON performing 32x16 multiplications via 16x16 multiplications, is more efficient than performing a 32x32 multiplication. Since multiplications consume a major part of the MCPS in an audio codec, a good reduction can be seen in overall MCPS using this.

Vectorizing Table Generation

The sin/cos coefficients required in IMDCT etc. are either stored in the form of tables or generated every time. Generating the sin/cos coefficients every time takes more cycles than accessing it from the tables. However, when the size of the table that needs to be stored is large, generation of sin/cos coefficients is preferred to reduce the memory footprint. Usually the sin/cos coefficients are generated using the following algorithm (or a similar algorithm).

Cos(a+b) = Cos(a-b) – 2 Sin(a) Sin(b)
Sin(a+b) = Sin(a-b) + 2 Cos(a) Sin(b)

After every iteration, the values are updated as shown below:
Cos(a-b) = Cos(a)
Sin(a-b) = Sin(a)
Cos(a) = Cos(a+b)
Sin(a) = Sin(a+b)

In the above algorithm, each sin/cos value is generated using the values generated from the previous iteration. Since each iteration is dependent on the result from the previous iteration, vectorization across multiple iterations is not possible. Though it is impossible to remove completely the dependency across iterations, it is possible to remove the dependency within a set of consecutive iterations so that they can be operated in the same vector.

The algorithm can be modified as shown below:
Sin (a+b) = Sin (a-b) – 2 Cos (a) Sin (b) → (Used in Iteration 1)
Sin (a+2b) = Sin (a-2b) – 2 Cos (a) Sin (2b) → (Used in Iteration 2)
Sin (a+3b) = Sin (a-3b) – 2 Cos (a) Sin (3b) → (Used in Iteration 3)
Sin (a+4b) = Sin (a-4b) – 2 Cos (a) Sin (4b) → (Used in Iteration 4)

After every four iterations, the values are updated as shown below:
Sin (a-4b) = Sin (a)
Sin (a-3b) = Sin (a+b)
Sin (a-2b) = Sin (a+2b)
Sin (a-b) = Sin (a+3b)
Sin (a) = Sin (a+4b)

In this way, the dependency is removed within sets of four consecutive iterations. The operations in these four iterations can be vectorized to occur in parallel.

16 bit Data Width

As discussed earlier maintaining the data in 16 bits will affect the audio quality. But, the data width can be reduced to 16 bits in some places in an audio codec without affecting the audio quality. Reducing the data width means that we can use 16x16 multiplications and hence a good reduction in the number of cycles. In decoders, the data width can be clipped to 16 bits before the last stage of processing which gives out the output PCM samples. For example in case of AAC decoder the output of IMDCT can be clipped to 16 bits, so that 16x16 multiplications can be used in the "overlap and add" module. In case of MP3 decoder the data can be clipped to 16 bits before the last stage of the synthesis filter bank, so that 16x16 multiplications can be used in the "Window and add" module in synthesis filterbank. Clipping the data width before last stage of the decoder does not affect the quality as the output data of that block is not processed further. In an encoder the data width can be kept in 16 bits for some of the Psychoacoustic blocks like "Transient detection", "Calculation of Band Energy ", where a 16 bits precision is sufficient to get a good audio quality.

Use of NEON Register Bank.

Higher Radix FFT algorithms

Usually FFT is used to implement the IMDCT/PQMF transforms optimally. It is important to choose an FFT algorithm that is most suited for NEON. In most audio codecs the transform length is usually a power of 2. Hence a radix-2 or a mixed radix algorithm (with radix-8, radix-4 etc. can are used). An algorithm with higher radix has an advantage of reduced number of computational operations. In addition, there is significant reduction in number of data loads/stores as we go for higher radix algorithms. However, higher the radix of the algorithm means more number of data points required for the computation one FFT butterfly.

So if sufficient number of registers is not available to hold all data points required for the computation of the butterfly then the data needs to pushed on the stack and popped back at a later point. This will offset the advantages of a higher radix algorithm. Since NEON has 32 registers (64-bit each), a higher radix FFT such as radix-8 or radix-16 becomes highly suitable for this platform. Hence, a mixed radix FFT algorithm with maximum number of radix-16 and radix-8 decimations will be the optimal implementation of FFT using NEON.

Merge Consecutive Functional Blocks

Merging of two or more consecutive functional blocks can be done in order to save loads/stores of data values. In most cases, the algorithm will allow the merger of two or more consecutive functional blocks, but often not done due to non-availability of registers (for storing data and the pointers). Since Cortex-A8 has enough registers, it is suggested to merge two functional blocks wherever possible and thus reducing the number of loads/stores.

Dual Issue

The NEON engine has dual issue capabilities i.e., two instructions can be executed in parallel. A load/store or a byte-permute type instruction (instructions for functions like swap, byte reverse, bit extract, bit pack/unpack) can be dual issued with a data-processing instruction (arithmetic and logical instructions), provided there is no data dependency between the two instructions. Bit pack/unpack instructions are used extensively in audio codecs, as it is required to unpack the 32-bit data into 16-bits for 32x16 multiplications. Byte reverse instructions are used in places when algorithm operates on an array in the reverse manner.

The normal structure of an audio processing loop is given below:

```
for i =1 to n iterations{
    Load the input data
    Rearrange the input data (byte permute instructions)
    Process the data (data processing instructions)
    Store the output data
}
```

The above loop it not suited for dual issue, as there is a data dependency between data processing and load/store, byte-permute instructions. Nevertheless, this loop can be restructured as shown below in order to make use of NEON's dual issue capabilities.

```
Load and rearrange the input data (for iteration 1)
Process the data (for iteration1)
Load and rearrange the input data (for iteration 2)
for i = 2 to n iterations
{
    Store the output data (for iteration i-1)
    Process the data (for iteration i)
    Load and rearrange the input data (for iteration i+1)
}
```

In above loop there is no data dependency between the load/store part of the loop and processing part of the loop, as the data operated by each of the block belong to different iterations. Hence the load/store, byte-permute instructions can be dual issued with data processing instructions to get a significant reduction in the number of cycles.

By using the above techniques, Ittiam has built a highly optimized MP3 Decoder on Cortex-A8 with Neon. The MP3 decoder is fully compliant to the ISO standards and consumes only 6.6 MCPS for decoding a 48kHz, 320 kbps stream.

Performance Results

With the optimized Neon implementation, an overall performance improvement of 1.7x to 2.1x against ARM11™ has been achieved for the audio decoder. The table below shows the performance gain achieved for various processing blocks in MP3 decoder.

Functional Block	Performance gain over ARM11 implementation
Huffman	1.1
Inverse Quantization	2.3
Inverse Channel transform (Mid-Side stereo, Intensity Stereo etc.)	3.5
IMDCT, Overlap-add	1.9
Syntheses filter bank (without including the Window and add stage)	2.0
Window and add stage of Syntheses filter bank	3.3

Conclusion

This paper discussed the various optimization techniques for implementation of Audio codecs on Cortex-A8 using the NEON technology. Optimization of Audio Codec on Cortex A8 involves lot of challenges in maintaining the precision/quality of the audio codec, while utilizing the NEON capabilities. Using the above optimization techniques, it is seen that overall performance of an Audio codec can improved by a factor of 1.7x to 2.1x with respect to an optimized ARM11 implementation.

END

References

[1] http://www.arm.com/iqonline/news/partnernews/23050.html
[2] Cortex-A8 Architecture Reference Manual
[3] RVCT v3.1 Assembler Guide