AMD'S 3DNow!™ VECTORIZATION FOR SIGNAL PROCESSING APPLICATIONS

Dongho Kim and Gwangwoo Choe

Advanced Micro Devices
5900 E. Ben White Blvd.
Austin, Texas 78741, USA

ABSTRACT

AMD's 3DNow!™ Technology provides substantial speedup for Digital Signal Processing applications. A set of DSP routines is vectorized with the 3DNow! ™ technology. The simplicity of the vector unit makes it easier to convert the conventional DSP programs into vector operations, thus reduces the learning curve. The performance gain from typical DSP routines such as FIR, IIR and FFT indicates that the speedup can reach up to 1.5 comparing to the conventional host-based signal processing units. 3D games and multimedia applications benefit from the technology. The vectorization can be integrated into compilers for the ease of use in increasing the performance of the signal processing applications.

1. INTRODUCTION

Multimedia applications consist of computational components commonly available from Digital Signal Processing (DSP) applications. Speed improvement of such computational elements has been investigated through a broad range of approaches such as Application Specific Integrated Circuits (ASICs), high-performance processors, and general-purpose microprocessor [1][2][3]. Current research focuses on improving the parallelism of the program in order to increase the performance. The ASIC design is cost effective, but lack of programmability. AMD's 3DNow!™ technology provides programmability as well as parallelism.

The 3DNow!™ technology provides a vector-processing unit in a general programming environment [4]. The vector units are organized such that they compensate the deficiency of Intel's MMX technology by increasing the performance on floating-point operations. The technology provides concurrent execution of two single-precision floating-point data with a throughput of an instruction cycle. The vector units are rather simple comparing to other parallel architectures, but it is required to convert the sequential programs into vector programs, so called vectorization [2][3].

Applications benefit from the technology if and only if they are vectorized. The size of the vector units determines the success rate of the vectorization. Due to the rather simple structure, the technology offers a high vectorization rate among its class. A loop either in single-nested or multiple-nested is vectorized when arrays are utilized in the program constructs. Conventionally, a complex vectorization is accomplished through techniques including loop distribution, partial vectorization, index-splitting, and subexpression vectorization [2].

In this paper, a set of DSP routines is analyzed to illustrate that the 3DNow!™ technology requires only a simple process for the vectorization. It has also shown that the performance analysis is close to the experiment result obtained from the execution of the routines on actual AMD-K6®-2 processor systems.

2. VECTORIZATION OF DSP MODULES

Digital signals have been processed in many different ways to support industrial applications such as multimedia applications. These applications are based on operations such as FIR filters, IIR filters and FFT [5]. These routines are not only fundamental in the applications but also provide important techniques applicable to many other operations. Vectorizing such operations illustrates how to convert digital signal-processing programs into a parallel program with vector operations

$$\bar{c} = \bar{a} + \bar{b} = \{a_U, a_L\} + \{b_U, b_L\} = \{a_U + b_U, a_L + b_L\}$$

where $\bar{a}$, $\bar{b}$, and $\bar{c}$ are vector operands. The 3DNow!™ technology provides multiplication, addition, subtraction, reduction, reciprocal, and comparison for the vector operation [4].

Reduction is an operation to accumulate both all the entries of the vector into a single floating-point data such as

$$c = c_U + c_L$$

The following notation will be utilized to present a loop over the multiplication of two variables.

$$\bar{y}(n) = \sigma(\bar{x}_{i^-} \cdot \bar{h}_{i^+})$$
$$= \sum_{i=0}^{m-1} \{x(n-2i), x(n-2i-1)\} \cdot \{h(2i), h(2i+1)\}$$

The subscript of the vector operands denotes the direction of the memory access either in forward or reverse direction as shown below.

$$\bar{x}_{i^+} = \{x(n-2i), x(n-2i+1)\}$$
$$\bar{x}_{i^-} = \{x(n-2i), x(n-2i-1)\}$$
$$\bar{h}_{i^+} = \{h(2i), h(2i+1)\}$$
$$\bar{h}_{i^-} = \{h(2i), h(2i-1)\}$$

In this sectioin, we will illustrate how simple the vectorization process is for the FIR, IIR and FFT routines with the vector notations.

2.1 FIR Filter Operation

The *k*-tap FIR filter operation is calculated by a summation

$$y(n) = \sum_{i=0}^{k-1} x(n-i)h(i)$$

Since each term of the summation is the multiplication of two variables, *x* and *h*, it can be grouped as a pair of multiplication.

$$y(n) = \sum_{i=0}^{m-1} \{x(n-2i)h(2i) + x(n-2i-1)h(2i+1)\}$$

where *m* is equal to a half of the tap size, *k*. Now this equation is programmed by the simple vector operation.

FIR-1: $y(n) = y(n)_U + y(n)_L, \bar{y}(n) = \sigma(\bar{x}_{i^-} \cdot \bar{h}_{i^+})$

It is also possible to change the index of variable *x* with that of variable *h*. This provides two additional vector programs.

FIR-1a: $y(n) = y(n)_U + y(n)_L, \bar{y}(n) = \sigma(\bar{x}_{i^+} \cdot \bar{h}_{i^-})$

FIR-2: $y(n) = y(n+1)_U + y(n-1)_L, \bar{y}(n) = \sigma(\bar{x}_{i^+} \cdot \bar{h}_{i^+})$

FIR-1 is a direct decomposition by swapping the input data or moving a pair of data into vector register in a reverse order. FIR-1a avoids the swap instructions by rearranging the filter coefficients in advance. FIR-2 is specially designed to compute the filter operation without either pre-arrangement of the filter coefficients or swapping the input data.

Since the swap operation of the input data adds extra more cycles, FIR-1 is less favorable than the other methods. However, FIR-1 is easy for compilers to produce the vectorization. FIR-1a is also simple but difficult for compilers to handle the filter coefficient in a global context. Programmers may rearrange the coefficients relatively easy, but it takes quite a bit of efforts for compilers to accomplish because the coefficients could be referenced throughout the entire programs.

2.2 IIR Filter Operation

The canonical form of IIR Filter is represented by

$$y(n) = b(0)x(n) + \sum_{i=1}^{N} a(i)y(n-i) + \sum_{i=1}^{N} b(i)x(n-i)$$

The summation can be converted into the following vector operations

IIR:

$$\bar{b}_0 = \{0, b(0)\},\ \bar{x}_0 = \{0, x(n)\}$$
$$\bar{y}(n) = \bar{b}_0 \bar{x}_0 + \sigma(\bar{a}_{i^+} \cdot \bar{y}_{i^-}) + \sigma(\bar{b}_{i^+} \cdot \bar{x}_{i^-})$$
$$y(n) = y(n)_U + y(n)_L$$

Since $y(n-1)\ldots y(0)$, $x(n-1)\ldots$, and $x(0)$ are the history buffer within a local function, they can be pre-arranged without affecting other programming components. Notice that the input signal $x(n)$ is directly loaded as the initial value of the vector register.

2.3 Fast Fourier Transform

Fast Fourier Transform algorithm is computed by applying a bufferfly addressing to the computational kernel,

$$X_{m+1}(p) = X_m(p) + X_m(q)$$
$$X_{m+1}(q) = (X_m(p) - X_m(q))W_N^r$$

where *p* and *q* are the index of the complex data *X* and *W*. Since the complex data requires both imaginary and real values, the vectorization is accomplished by converting X_m, X_{m+1}, and W_N^r into $\bar{x}$, $\bar{y}$, and $\bar{w}$, respectively.

FFT:

$$\bar{y}(p) = \bar{x}(p) + \bar{x}(q)$$
$$\bar{t} = \{t_U, t_L\} = \bar{x}(p) - \bar{x}(q)$$
$$\bar{y}(q) = \{t_U w_U - t_L w_L, t_U w_L + t_L w_U\}$$

An *N*-point FFT uses log*N* stages where the kernel is computed for each node and the results of the final stage are located at the memory in the butterfly-addressing mode.

Converting the complex data into the vector provides a simple vectorization for the FFT routine. However, it may take substantial amount of efforts for compilers to produce such vectorization because the compiler needs to perform code-moving [6] in order to create the complex data type from the float data type.

3. PERFORMANCE ANALYSIS

The vectorized version of the DSP program runs faster than conventional program. However, the performance gain is not identical to the theoretical upper bound that is the same as the number of vector unit. For example, 2-way vector does not mean that it can provide a speedup of 2, due to the presence of non-vector portion of the program. A fully vectorized version is referred to as ideal case, and the others as an actual execution in this paper.

3.1 FIR

Since the conventional implementation of the *k*-tap FIR filter utilizes a scalar operation by multiplying and adding each term one at a time, the total instruction counts are

k multiplication + (*k-1*) *addition*
(*k-1*) *delays*

The sequential execution of such implementation by utilizing individual arithmetic takes a total of

$$T_1 = kT_m + (k-1)T_a + (k-1)T_d$$

where T_m, T_a, and T_d are the execution time of the multiplier, adder and delay instruction, respectively.

FIR1 reduces the total number of instruction in half along with swap instructions.

$\lceil k/2 \rceil$ *multiplication* + $\lceil k/2+1/2 \rceil$ *addition*
(*k-1*) *delays*
$\lceil k/2 \rceil$ *swaps*
1 reduction

The latency of executing the program is

$$T_2 = \left\lceil \frac{k}{2} \right\rceil T_m + \left\lceil \frac{k+1}{2} \right\rceil T_a + (k-1)T_d + \left\lceil \frac{k}{2} \right\rceil T_s$$

FIR2 provides better performnace due to the elimination of the swap operations. Thus, the total latency is

$$T_3 = \left\lceil \frac{k}{2} \right\rceil T_m + \left\lceil \frac{k-1}{2} \right\rceil T_a + (k+2)T_d$$

The FIR1 and FIR2 provides the following speedup,

$$S_3 = \frac{T_1}{T_3} = \frac{kT_m + (k-1)T_a + (k-1)T_d}{\frac{k}{2}T_m + (\frac{k-1}{2})T_a + (k+2)T_d}$$

$$S_2 = \frac{T_1}{T_2} = \frac{kT_m + (k-1)T_a + (k-1)T_d}{\frac{k}{2}T_m + (\frac{k-1}{2}+1)T_a + (k-1)T_d + \frac{k}{2}T_s}$$

This illustrates that the speedup is not only the function of the execution time of the multipliers and adders, but also the delays. If the delays (or moving data from memory) take relatively long comparing to the multipliers and adders, then there will be very little speedup from the vectorization.

3.2 IIR

Since the IIR filter operation is based on cascading the multiple copies of the canonical IIR kernels, the performance can be measured by comparing the delay of the single kernel. Suppose the kernel has *N* history buffers, then the kernel requires

(1+2*N*) *multiplication* + 2*N addition*
N delays

The sequential execution of the kernel takes a total of

$$T_1 = (1+2N)T_m + 2NT_a + NT_d$$

The vectorization reduces the total delay to

$$T_2 = (1+N)T_m + NT_d + 2T_a$$

Therefore the speedup is

$$S = \frac{T_2}{T_1} = \frac{(1+N)T_m + NT_d + 2T_a}{(1+2N)T_m + 2NT_a + NT_d}$$

and it is a function of the execution time of the multiplier, adder, and move instructions.

3.3 FFT

The entire process of FFT routine repeats the butterfly kernel *N* times per each stage. There are a total of $\log_2 N$ stages to complete the *N*-point FFT. Thus, the total number of instructions of sequential program is

$$N \log_2 N \ \textit{addition} + N \log_2 N \ \textit{subtract}$$

$$\left\lceil \frac{N}{2} \right\rceil (\log_2 N - 1) \ \textit{complex multiplication}$$

The vectorization reduce it to

$$\frac{N}{2} \log_2 N \ \textit{addition} + \frac{N}{2} \log_2 N \ \textit{subtract}$$

$$\left\lceil \frac{N}{2} \right\rceil (\log_2 N - 1) \ \textit{complex multiplication}$$

If we consider the subtract takes the same as the addition, then the execution time of the sequential program is

$$T_1 = 2N \log_2 N \cdot T_a + \frac{N}{2} T_c$$

where T_a and T_c are the execution time of adder and complex multiplier, respectively.

The vectorization reduces the total delay to

$$T_2 = N \log_2 N \cdot T_a + \frac{N}{2} T_c$$

Thus, the speedup is

$$S = \frac{T_1}{T_2} = \frac{2N \log_2 N \cdot T_a + \frac{N}{2} T_c}{N \log_2 N \cdot T_a + \frac{N}{2} T_c}$$

It indicates that the speedup is a function of the execution time of adders and complex multiplier.

4. EXPERIMENTAL RESULTS

A set of experiments has been conducted in order to measure the performance of the vectorization both in a practical operating environment and ideal benchmark cases. We have chosen programmers who are skilled in digital signal processing applications, but relatively poor in programming language as well as the 3DNow! ™ technology. The skill set is to demonstrate the learning curve in adapting their applications to the new technology.

The novice programmers were asked to program the DSP modules and convert them into the vector operations by utilizing the in-line assembly of C/C++ programming language. It took them about a week in the conversion after two months of work in developing a series of demonstration programs as well as the benchmarks. Then, we run the DSP routines on AMD-K6®-2 300Mhz microprocessor to measure the performance.

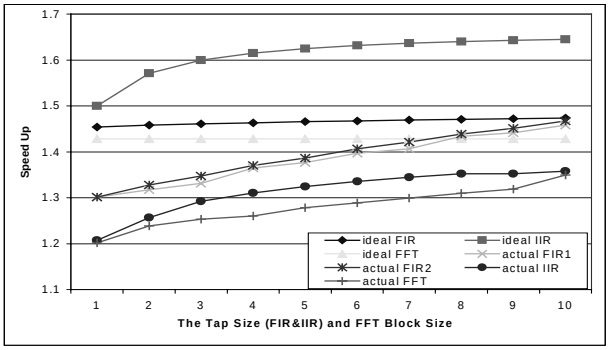

Figure 1- Performance Result

Figure 1 shows the result of the performance measurement. The ideal case is based on the analysis examined earlier, and the actual cases are obtained from running the equivalent programs. The tap size of the IIR filters indicates the total number of filter coefficients. The FIR filters utilize taps that are the multiple of the figure indicated. For example, 1 means 20 taps and 10 is 50 taps for the FIR filters. The block size of FFT routine indicates the stage of the FFT routine, therefore the block size, *n*, means 2^n-point FFT routine.

The ideal speedup is close to 1.5. On the other hand, the actual speedup turns out to be approximately 1.3. We have discovered that the discrepancy is due to the overhead caused by the in-line assembly of the C/C++ as well as the data movement from the memory to vector register that we didn't take into the consideration for the ideal cases. This illustrates that the speedup is between 1.3 and 1.5 regardless of the methodology.

We have also applied the FIR-1a routine to wavelet transforms with 9-tap FIR filters introduced by Shapiro [7]. Profiling of the analysis and synthesis filters of the wavelet transforms indicates that more than 40% of the total execution is still occupied by non-filter operations as shown in Table 1.

Table 1 - Profiling of Wavelet Transforms

	Analysis Transforms	Synthesis Transforms
total	3037.883	2324.189
filters	1771.544	1259.198
%	58.32%	54.18%

Two separate optimizations were attempted: vector move and vectorized FIR filter. The vector move replaces all the move instructions with the 3DNow!™ instructions, and the other is to plug in the previously developed FIR routine into the wavelet transforms. Table 2 shows the result of performing the wavelet transforms: analysis filter first, and then synthesis filter on Mandrill image of size 512x512 with 6 level transforms [8]. It indicates that the speedup reaches up to 1.5 even if a half of the program is not vectorizeable.

Table 2 - Speedup of Wavelet Transforms

	Analysis-Synthesis	Speedup
Reference	427643256	1.00
Vector Move	365428457	1.17
Vectorization	336622616	1.27
Both of them	280416549	1.53

Mandrill image has been restored from EZW algorithm [7][8]. Figure 2 shows the image quality from the compression process. The difference image is gray level indicating $127+16\cdot(x-\hat{x})$ where x and $\hat{x}$ are the original image and the restored, respectively. A test pattern is also included for the intensity level of $-8 \le (x-\hat{x}) \le 8$. The objective measurement indicates that the restored image is 45.28dB PSNR compared to the original image while the compression ratio is still above the unity [8].

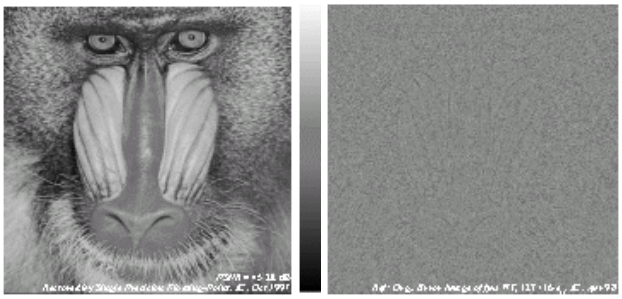

Figure 2 - Image Quality of 3DNow!™ Technology

5. CONCLUSION

A set of DSP routines is vectorized with the 3DNow!™ technology. The vectorization increases the performance of DSP routines with a speedup ranging 1.3 through 1.5. Our experiment indicates that the programming language heavily affects the performance. The in-line assembly of C/C++ introduces an overhead, but it makes the vectorization process substantially simpler by allowing free accesses to C/C++ variables.

We have also applied the same techniques to actual multimedia applications such as wavelet transforms. This experiment indicates that the 3DNow!™ technology increases the speedup up to 1.5 even if there are substantial portions of the program not vectorized. The 3DNow!™ technology is relatively easy to program due to the simplicity of the architecture as well as the small number of instruction set.

6. ACKNOWLEGEMENTS

The authors would like to thank Ned Finkle and Jim MacDonald for their support at Advanced Micro Devices.

7. REFERENCES

[1] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontroller," *IEEE Design & Test of Computers*, December 1993, pp. 1199-1208.

[2] I. M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.

[3] V. Zivojnovic, S. Ritz, and H. Meyr, "Retiming of DSP programs for optimum vectorization," *Proceedings of the ICASSP'94*, vol. 2, 1994, pp. 465-468.

[4] H. Kalish and J. Isaac, *The AMD-K6 3D Processor*, Abacus, 1998.

[5] A. V. Oppenheim, and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tool*s, Addison-Wesley, 1987.

[7] J. Shapiro, "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Trans. Signal Processing*, vol. 41, 1993, pp. 3445-3462.

[8] G. Choe and E. E. Swartzlander, Jr. "Merged Arithmetic for Computing Wavelet Transforms," *Proceedings of the 8th Great Lakes Symposium on VLSI*, 1998, pp. 196-201.