SOME FAST SPEECH PROCESSING ALGORITHMS USING ALTIVEC TECHNOLOGY

Sanjay M. Joshi *

CSEE Dept, University of Maryland
Baltimore County, Baltimore, MD 21250,
USA. Email: joshi@ieee.org

Pradeep K. Dubey

IBM Research Division
Solutions Research Center, New Delhi,
India. Email: pkdubey@in.ibm.com

ABSTRACT

The AltiVec technology is a SIMD (Single Instruction Multiple Data) extension to PowerPC architecture. It is intended to provide architectural support for performance improvement of various image and signal processing applications, including speech processing, on a general-purpose processor implementation, such as, the PowerPC line of processors. In this paper we have implemented some of the common speech processing algorithms on AltiVec architecture. The algorithms discussed in this paper are autocorrelation computation, linear prediction coefficients computation via Levinson-Durbin method and Schur recursion, and part of the GSM speech compression system. AltiVec obtained significant speedups on all these algorithms, compared to the scalar PowerPC implementation. We also found that additional speedup was achievable by porting to new, more SIMD-friendly algorithm.

1. INTRODUCTION

Motorola recently announced the AltiVec technology, a vector extension consisting of 162 new instructions, to the PowerPC architecture [1] [6, 7]. The AltiVec technology is very general-purpose in nature. As a result, it is applicable across a wide range of applications. It applies wherever there is data parallelism to be exploited. This paper deals with optimizing some of the basic operations in speech processing using AltiVec technology. The operations optimized are autocorrelation and linear prediction coefficients computation. As a practical example, we optimized a part of the GSM compression system. The outline of this paper is as follows. The following section is a brief overview of the AltiVec technology. The next 3 sections discuss the optimizations. The conclusions are drawn in the last section.

* Corresponding author

This work was done at IBM T. J. Watson Research Center, NY

[1] AltiVec is a trademark of Motorola, Inc. PowerPC and PowerPC architecture are trademarks of International Business Machines Corporation.

2. ALTIVEC TECHNOLOGY

The AltiVec technology offers 32 vector registers, each 128-bits wide. These registers can be split as sixteen 8-bits integers, eight 16-bits integers, four 32-bits integers, or four 32-bits IEEE single-precision floating point data elements. These vector registers are in addition to, and architecturally separate from the integer and floating point registers of PowerPC. All AltiVec instructions operate on fixed-length, 128-bit vectors, each performing the same operation on corresponding elements in the source vector operands. Apart from the intra-element operations, AltiVec also provides inter-element operations, such as sum of products and sum-across. One of the most powerful inter-element operations is the permute operation, which allows arbitrary selection of up to 16 byte elements from a set of 32 bytes. This operation performs two very important functions, namely, data-reorganization and table lookup. AltiVec also provides the 3-source operand form of multiply accumulate, multiply-add. Multiply-add primitive multiplies the respective elements of two source vector registers, and adds the intermediate results to the corresponding elements of a third source register (the accumulator). The result is deposited in the target vector register. Architectural support is also provided for vector dot product. This is accomplished with two instructions: multiply-sum and sum-across. The multiply-sum primitive multiplies corresponding elements in two source vector registers and sums the adjacent elements along with corresponding value from a third source register into four 32-bit partial sums, deposited in the target vector register. Finally, note that a fairly large amount of data can be stored in the 32 on-chip vector registers. This improves performance in many cases by avoiding memory spills of intermediate values.

We implemented some of the common speech processing algorithms in C and AltiVec assembly. The code generated by the C compiler for the Scalar PowerPC instruction set was found to be quite optimal. For the AltiVec, compiler development is still incomplete. We, therefore, coded the AltiVec routines in assembly and invoked them from the C program. The performance analysis presented in this pa-

per is based on a cycle accurate simulation of a possible superscalar implementation of PowerPC architecture with AltiVec extensions. Important micro-architectural characteristics of this implementation are as follows:

- All instructions fully pipelined with single-cycle throughput, simple ops: 1-cycle latency, compound ops (those involving multiply): 3-4 cycle latency.
- Dual AltiVec instruction issue: one arithmetic, one *permute* class instruction (*permute* class refers to instructions involving inter-element rearrangements, like pack, unpack, permute operations)
- No restriction on issue with scalar PowerPC instructions

The following sections describe the results of our work.

3. AUTOCORRELATION COMPUTATION

Autocorrelation computation is one of the first steps in computing linear prediction coefficients (LPCs). It is also useful for many other DSP applications. We investigated autocorrelation computation from the speech processing point of view, where only first few autocorrelation coefficients are needed. If all the coefficients are needed, they can be computed as the magnitude of the Fourier transform, which can be computed very efficiently using FFT algorithms.

Autocorrelation coefficients $r(k)$ of a signal $s(k)$ are defined as $r(k) = \sum_{n=0}^{N-1-k} s(n)s(n+k)$, where N is the length of the signal and $0 \leq k < N$. The signal is assumed to be real, which is the most common case in speech processing. To compute an order p linear prediction filter, $r(0)$ through $r(p)$ are needed.

One of the most important considerations is the size of each sample of the signal. If signal samples have fewer number of bits, the whole computation or part of it can be computed as integer operations without overflows. We assumed that the samples are stored as 8-bit unsigned quantities and the actual sample values are computed by multiplying them with a constant and adding an offset to keep the algorithm general in nature.

If a signal is scaled, or multiplied by a factor, the linear prediction filter does not change. Adding an offset, however, changes the filter. Let the constant offset be t. The autocorrelation coefficients $r_s(k)$, then, can be written as

$$\begin{aligned} r_s(k) &= \sum_{n=0}^{N-1-k} [s(n)+t]\,[s(n+k)+t] \\ &= r(k) + t\left[2\sum_{n=0}^{N-1} s(n) - \sum_{n=0}^{k} s(n) - \sum_{n=N-1-k}^{N-1} s(n)\right] + t^2(N-k) \end{aligned}$$

The computation of $r(k)$ uses only unsigned 8-bit quantities and for $N = 256$, the result can be stored in a 32-bit integer without any overflows. Similarly, the results of the sums in the second term can also be stored in a 32-bit integer.

The above algorithm was implemented in C and in AltiVec assembly. The results are given in Table 1. The speed-up was calculated as the ratio of number of cycles for scalar PowerPC implementation to the number of cycles for AltiVec implementation.

Order	4	8	12	16
PowerPC (Cycles)	5004	9292	13907	20779
AltiVec (Cycles)	276	407	543	676
Speedup	18.13	22.83	25.61	30.74

Table 1: Performance comparison for autocorrelation computation.

LPC computation usually needs no more than 16 autocorrelation coefficients. Hence we could store all the autocorrelation coefficients in the large registers space available. The C version used individual variables for computations rather than arrays, except for the data samples, so that the code generated by the compiler made optimum use of scalar PowerPC registers.

The large speedups are obtained because we can perform 16 multiplications and 16 additions in one instruction, as opposed to 32 instructions to do the same in scalar PowerPC.

4. LINEAR PREDICTION

Linear prediction coefficients (LPCs) is one of the most common techniques used in speech analysis [2]. It forms one of the building blocks of speech compression systems. Typically the computation of LPCs is carried out for 12 to 16 coefficients. Two popular methods for LPC computation which use the autocorrelation coefficients $r(k)$ of a real signal $s(m)$ were implemented on AltiVec for performance comparison.

4.1. Levinson-Durbin algorithm

In the Levinson-Durbin (LD) recursion algorithm [5], the LPC filter is started with order 1 (which is $a_0^0 = 1$), and is grown recursively using reflection coefficients K_i. The algorithm is very efficient on a scalar processor like PowerPC. On a parallel processor, however, the computation of reflection coefficients K_i, is a bottleneck [4]. Computation

of K_i involves inverting the coefficient vector, which is a costly operation. We, therefore, store an inverted copy of the LPC vector. It can be used in growing the LPC vector later. The resulting algorithm is listed below.

1. $E^0 = r(0), a_0^0 = 1, b_0^0 = -1.$
2. for $i = 1 : p, K_i = \left[\sum_{j=1}^{i} b_{j-1}^{i-1} r(j)\right] / E^{i-1}$
3. $\quad b_0^i = -K_i$
4. $\quad$ for $j = 1 : i, b_j^i = b_{j-1}^{i-1} + K_i a_j^{i-1}$
5. $\quad\quad a_j^i = a_j^{i-1} + K_i b_{j-1}^{i-1}$
6. $\quad E^i = (1 - K_i^2) E^{i-1}$

It can be easily verified that the reflection coefficient K_i in this algorithm is the sign reversed K_i from the standard LD algorithm.

4.2. Schur recursion

Kung and Hu [4] proposed a parallel algorithm based on Schur method [8], used for solving a Toeplitz matrix. In Schur recursion (SR), the system is solved by triangularization or LU-decomposition of the autocorrelation matrix. Their algorithm, simplified for real data and expanded to compute LPCs from reflection coefficients z, is given below as a C code.

```
for(i=-p; i<= 0; i++)
   u[-i] = v[-i] = r[-i];

a[0] = b[0] = 1;
for(i=1; i<=p; i++)
{
   z = -v[i]/u[i-1];
   for(k=-p; k<-i+1; k++)
   {
      u[-k] = u[-k-1] + z*v[-k];
      v[-k] = v[-k] + z*u[-k-1];
   }
   for(k=i; k>0; k--)
   {
      b[k] = b[k-1] + z*a[k];
      a[k] = a[k] + z*b[k-1];
   }
   b[0] = z;
} /* for i */
```

4.3. Results and discussion

The standard LD algorithm was implemented in C. The modified LD algorithm was implemented in AltiVec assembly. The SR algorithm was implemented both in C and AltiVec assembly.

A 256-point random data was generated using MATLAB and filtered using a finite impulse response filter. The autocorrelation coefficients were computed using MATLAB and used as input to LD and SR algorithms in 32 bits floating point format. The results of both the algorithms were checked against those obtained using the `levinson` function of MATLAB. The timing results are presented in Table 2. The overall speedup was computed as the ratio of the number of cycles for the best algorithm on scalar PowerPC and that for the best algorithm on AltiVec.

Order	4	8	12	16
LD on PowerPC (cycles)	253	639	1225	1947
LD on AltiVec (cycles)	102	234	388	569
SR on PowerPC (cycles)	306	860	1638	2784
SR on AltiVec (cycles)	64	142	238	366
Speedup for LD	2.48	2.73	3.16	3.42
Speedup for SR	4.78	6.06	7.08	7.61
Overall speedup	3.95	4.50	5.15	5.32

Table 2: Performance comparison between for LPC.

The main reasons for the speedups are as follows. For the LD algorithm, a speedup of 4 is achieved in computing the numerator of K_i and updating a. But updating b adds more computations. The load/store instructions in the iterations are saved since all the variables can be stored in AltiVec registers.

For the SR algorithm, all the steps except the computation of K_i are speeded up by a factor of 4. Further, all the load/save instructions are not necessary because all the arrays can be stored in registers. Therefore the speedup is significant.

It should be noted that the results on scalar PowerPC and AltiVec were not exactly the same. The reason is that we used a reciprocal estimate in division on AltiVec, while the scalar PowerPC uses actual floating point division. The error was usually in the fourth or fifth decimal places in the coefficients.

The LD algorithm is very efficient on a scalar processor. The SR algorithm does not work as fast as the LD algorithm on scalar PowerPC. On AltiVec, however, the SR algorithm performs much better than any of the two algorithms on scalar PowerPC.

5. GSM SPEECH COMPRESSION

As an example of practical example, we implemented a section of speech compression algorithm used in the global sys-

tem for mobile communication (GSM), Europe's currently most popular protocol for digital cellular phones.

The `toast` library [1] was downloaded. After profiling using the `gprof` program, it was found that about 39.8% of computation time was consumed by long-term prediction (LTP) coefficient computations. So we concentrated on speeding up the LTP computations, corresponding to module number 4.2.11 in the GSM recommendation [3].

5.1. LTP computations

The LTPs are basically cross-correlation coefficients between two signals. The output of the function is the highest coefficient and its location. The signals are 40 and 120 samples long. There are 80 coefficients totally. Each sample of the signal is a 16-bit signed integer. The LTPs are 32-bit integers.

Since 8 samples of the signal could be fitted in one AltiVec register, both the signals could be fitted in the registers. The longer signal was then shifted by 1 sample and the LTPs were computed. The highest value and its position was retained.

The scalar PowerPC program took 12941 cycles for executing the LTP function once. The AltiVec version gave the same results in only 1034 cycles, a speedup of 12.5. The main reasons for this speedup are that AltiVec does 8 multiplications and 8 additions in 1 instruction, as opposed to 16 to do the same on scalar PowerPC. Assume it takes $100x$ seconds for executing the GSM compression program, and $39.8x$ seconds for computing the LTP parameters. In that case, AltiVec will take only $3.2x$ seconds for LTP parameter computations. Thus, even without optimizing any other function, the compression is done in only $63.4x$ seconds.

6. CONCLUSIONS

In this study, we implemented some of the commonly used DSP algorithms on the AltiVec architecture. We compared the execution times with those for the scalar PowerPC processor. It was seen that AltiVec significantly increases the speed of execution.

The manner in which the 128-bit wide AltiVec registers are split is important. If the data is 32-bit, 4 samples can be processed simultaneously. For a 16-bit or 8-bit data, even more parallelism can be obtained. Traditionally, the algorithms have been choosing the highest resolution available (often 32 bits), because no significant advantage was possible by processing at lesser resolution, except while storing. As shown by the autocorrelation computations, if the size of each data sample is small, AltiVec can speed up the computations by a large factor.

Many popular algorithms were developed with scalar processors in mind. In these algorithms the goal was to minimize the number of computations. With the AltiVec architecture, however, proper alignment of data can play a significant role by exploiting parallelism. An algorithm may work faster if it has the right kind of parallelism, even though it may require more scalar computations. This was demonstrated by computation of linear prediction coefficients. The Schur recursion outperformed the popular Levinson-Durbin algorithm, even though the latter had fewer computations.

We implemented a part of the GSM speech compression system to demonstrate the capabilities of AltiVec. Some of the standard algorithms were designed with the computational costs of various functions in mind. The AltiVec offers a faster implementation now. So some of the methods that were computationally too costly earlier, can now be affordable. The AltiVec technology, thus, can play a vital role in designing new DSP applications, or modifying the existing standard algorithms.

In this paper we were mainly interested in implementing some basic DSP algorithms. We wanted to demonstrate the tremendous potential of the AltiVec architecture beyond the easily parallelizable multimedia applications. At this point many directions lie open for further work. The basic algorithms developed here can be combined to speedup a standard application, like speech coding. Many more basic algorithms can investigated for underlying parallelism and a library of several functions can be developed. A standard algorithm can be profiled, and the most time-consuming functions can be optimized using AltiVec.

7. REFERENCES

[1] J. Degener, "toast library," ftp://ftp.cs.tu-berlin.de/pub/local/kbs/tubmik/gsm/, Technical University of Berlin FTP server, Jul 5 1995.

[2] J. R. Deller, Jr., J. G. Proakis, and J. H. L. Hansen, *Discrete-Time Processing of Speech Signals*, New York: MacMillan, 1993.

[3] ETSI, *GSM Full Rate Speech Encoding*, Recommendation 06.10, European Telecommunications Standards Institute, B.P.152, F-06561 Valbonne Cedex, France, 1990.

[4] S.-Y. Kung and Y. H. Hu, "A highly concurrent algorithm and pipelined architecture for solving Toeplitz systems," *IEEE Trans. Acoust. Speech Signal Processing*, Vol. ASSP-31, No. 1, pp. 66-76, Feb. 1983.

[5] J. Makhoul, "Linear prediction: A tutorial review," *Proc. IEEE*, Vol. 63, pp. 561-580, 1975.

[6] Motorola, Inc, "Motorola AltiVec Technology," http://www.mot.com/SPS/PowerPC/AltiVec/.

[7] M. Phillip, K. Diefendorff, P. Dubey, R. Hochsprung, B. Olsson, and H. Scales, "AltiVec (tm) technology: Accelerating Media Processing Across the Spectrum," HotChips 10, Stanford University, California, August 16-18, 1998.

[8] I. Schur, "Uber potenzreihen die in innern des einheitskreises beschrankt sind," *J. Reine Angewandte Mathematik*, Vol. 147, pp. 205-232, 1917.