Native Signal Processing With Altivec In the Ptolemy Environment
Ken Aponte and Ken Logan
May 10, 2000

Abstract

The authors extend the functionality of the Ptolemy simulation and code generation facilities by implementing Altivec enabled signal processing kernels, the FFT and FIR, as Ptolemy actors. These actors are then used in Ptolemy demo systems, compiled using the GNU compiler enhanced with support for Altivec, and simulated using the Psim PowerPC simulator. This integration of tools, with Ptolemy as the foundation, provides the design, synthesis and simulation environment necessary for rapid prototyping of systems. Performance results are obtained and presented for all tests utilizing both scalar and Altivec versions of the actors.

1 Introduction to Native Signal Processing

Many of the modern general-purpose processor architectures now include native signal processing (NSP) extensions. Hewlett Packard PA-RISC, Sun Sparc, Silicon Graphics MIPS, Digital Alpha, Intel x86, AMD x86, and Motorola PowerPC have all introduced single instruction multiple data (SIMD) instruction set extensions to take advantage of the data parallelism inherent in streaming signal processing and graphics applications. There is a great deal of diversity in the features included in the various NSP extensions, possibly due to the fact that we are just beginning to understand the workloads targeted by the extensions.

Altivec is unique among the NSP extensions because it adds support for a separate 128 bit vector multimedia unit in the processor [1]. Altivec technology is available in the currently shipping PowerPC 7400 processor and another recently announced [2] member of the PowerPC G4 family.

Altivec is the only NSP extension that we identified through our research which offers thirty-two 128 bit wide dedicated vector registers. Unlike the Intel x86 compatible NSP extensions, there is no performance penalty associated with a ‘context switch’ to switch in or out of vector mode. In fact, it is possible to write code that uses the integer unit, vector unit, and floating point unit concurrently on a PowerPC processor that implements Altivec [3]. The vector registers provide 8-way parallelism for 16-bit signed and unsigned integers and 16-way parallelism for 8-bit signed and unsigned integers. Saturation arithmetic and a rich variety of

instructions are included in the Altivec instruction set. The Altivec programming model is documented in [4,5].

Currently, programmers must modify existing applications or write applications explicitly for a certain NSP extension. In [6, 7, 8] it was stated that this requirement presents significant usage problems for the NSP extensions. Since the new data-types and operations on them are not standardized, code utilizing them is not portable at the source level between different NSP extensions. Using libraries provided by the processor vendors doesn't remedy this portability problem, due to the fact that the library API's are also not standardized. Compiler support typically uses function inlining or macro calls in code segments that will benefit from using the NSP extensions [6]. Ideally, compilers of the future will be able to analyze and 'auto-vectorize' code to be optimized for a given NSP extension. This would make the new semantics invisible to the programmer, but such compilers do not currently exist [6]. A possible solution to this problem is finding an abstract representation that can efficiently be converted to software for an arbitrary NSP extension. We discuss using Ptolemy to accomplish this solution in the next section.

2 Programming with Altivec in Ptolemy

Ptolemy is a software environment for the simulation and prototyping of heterogeneous systems [9]. It provides a software engineer or hardware designer a clear view of natural partitions of software and hardware in a single, heterogeneous environment.

This is accomplished in Ptolemy by providing an object-oriented kernel that is free from any particular model of computation. New models of computation (domains) can be easily added without affecting existing domains. A domain may either simulate on a desktop workstation or synthesize code. Once implemented, domains can be interwoven and manipulated. Thus, Ptolemy provides a heuristic approach to specify, simulate, and synthesize heterogeneous systems, which in general, is a very difficult problem.

"Code generation" refers to the synthesis of software corresponding to the algorithm [9]. The Ptolemy stars we developed produce C source code that use Motorola's Altivec extensions. Ptolemy (combined with the tools described below in section 3.1) provides a complete design

system from concept to implementation to synthesis and test. The kernels developed by the authors can form the basis of larger systems. Such systems can be prototyped with relative ease in a "plug-and-play" manner – Altivec stars are substituted for their scalar versions in a graphical environment and almost immediately the resultant system can be evaluated. Chen, Reekie, Bhavem, and Lee conducted similar work in [7] using the NSP instructions of the Sun UltraSparc architecture, VIS.

3 Implementing Altivec enhanced NSP Kernels

3.1 Tool-set used

We used several tools in addition to Ptolemy to complete our experiments. The GNU compiler enhanced with support for Altivec [10] was used to compile the code generated by Ptolemy. The GNU compiler was configured as a cross-compiler targeting 'powerpc-eabisim.' The 'powerpc-eabisim' executables were then run on the Psim PowerPC simulator. The Psim simulator is included among the GNU tools bundled with the GNU compiler. We used the trace generation capabilities of Psim to generate traces that were then used by the sim_g4 [11] PowerPC timing simulator to provide detailed timing information.

3.2 Fixed Point FIR Implementation

We chose to implement the FIR filter due to its widespread use in signal processing applications. It was shown by Daubechies in [12] that under certain regularity conditions discrete-time filters will lead to continuous-time wavelets. This is a very practical and extremely useful wavelet decomposition scheme, since FIR discrete-time filters can be used to implement them. Wavelet transforms have gained widespread acceptance in signal processing in general, and in image compression research in particular.

Another contemporary example of the use of an FIR is in computer graphics. Aliasing effects in computer-generated images are seen in the jagged edges of rendered objects. Anti-aliasing is a process used by computer graphics applications where pixels are the sum of N nearest neighbors. This "smooths" the edges of objects by making the edge transitions gradual. Another benefit of this method is that transient noise or "speckles" are filtered from the image. An FIR filter is ideally suited for this task.

An FIR filter works by multiplying an array of the most recent n data samples by an array of constants (called the tap coefficients), and summing the elements of the resulting array. (This operation is commonly called a dot product.) The filter then inputs another sample of data (which causes the oldest piece of data to be thrown away) and repeats the process.

The formula for an N tap FIR is $Y(n) = \sum_{k=0} X(n-k)H(k)$ [EQN. 1] where Y(n) is the filtered value at time equal n and X(n-k) is the sequence of the last N input values and H(k) is the set of discrete filter values. The result is an output Y that is a weighted sum of the last N values of X.

Figure 1 at right shows the vector representation of EQN 1. The figure shows the parallelism inherent in the algorithm alongside the Altivec operations utilized in the kernel.

Using a vector of 8 signed integers, a 16-tap FIR can be unrolled by a factor of 8:1 (a 16 cycle loop in the scalar implementation can be reduced to a 2 cycle loop in the Altivec implementation).

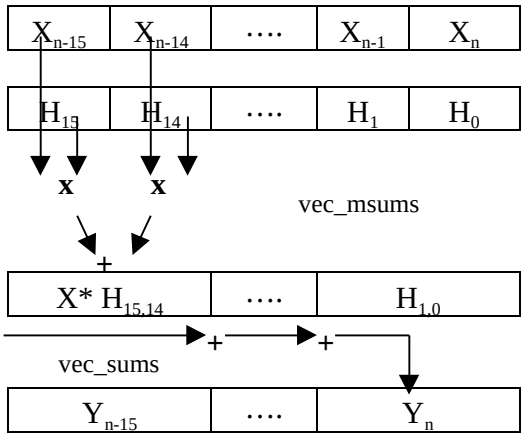

Figure 1 FIR Algorithm Diagram

3.3 Fixed Point FFT Implementation

The FFT is widely used in signal processing applications, for applications ranging from spectrum analysis, to sonar beamforming, to engine knock-detection in automotive applications. We chose to work with a 16 bit fixed-point, radix-2, decimation in frequency, complex Fast Fourier Transform (FFT) algorithm. Although Altivec includes support for floating point arithmetic, we chose a fixed-point implementation so that a comparison to the FFT implementation from [7] is possible.

Although it is beyond the scope of this paper to discuss the theory behind the FFT algorithm, the details of the particular FFT algorithm that we implemented provide a good insight into signal processing with Altivec. The primitive operation of a FFT is the 'butterfly.' A butterfly requires one complex addition, one complex subtraction, and one complex multiplication operation. Butterflies are performed on different combinations of the input data-

points by iterating over the input data with different stride lengths. For a 2^N point FFT, it is necessary to perform N passes over the input data. The first N-2 passes are implemented in a nested loop that is three levels deep. The butterflies are performed in the inner loop. Pseudo-code for one butterfly operation is as follows:

```
1) RETMP   = REIN[X] - REIN[Y];
2) IMTMP   = IMIN[X] - IMIN[Y];
3) REIN[X] = REIN[X] + REIN[Y];
4) IMIN[X] = IMIN[X] + IMIN[Y];
5) REIN[Y] = ((RETMP * Tw0) - (IMTMP * Tw1)) SHIFTRIGHT scalelog2;
6) IMIN[Y] = ((RETMP * Tw1) + (IMTMP * Tw0)) SHIFTRIGHT scalelog2;
```

Except for the multiplications, which have a 32-bit result, all the values involved are 16-bits wide. Since fixed-point arithmetic is being performed, the result of the multiplications must be shifted right in order to maintain position of the decimal point. The strategy we used to obtain a speedup in the Altivec version is to compute 4 butterflies at a time in the inner loop. We implemented the scalar FFT to also perform 4 butterflies in its inner loop in order to keep the comparison fair to both versions. A separate Ptolemy star provides input to the Altivec FFT star in the format:

Real0	Imag0	Real1	Imag1	Real2	Imag2	Real3	Imag3

Lines 1-2 of the above pseudo-code are achieved for 4 butterflies at a time simply by doing an Altivec vector subtract of signed short type vectors. Lines 3-4 are similarly implemented, only with vector addition. Lines 5-6 require more effort. Since the Altivec ISA doesn't provide a fixed-point multiply instruction, we are forced to use a form of multiply that accepts 16 bit operands and produces a 32 bit result which then is shifted according to the scale factor using a separate vector operation. The result is that the inner loop requires four vector multiply instructions to perform the sixteen fixed-point multiply operations necessary to compute four FFT butterflies.

A different operation is performed in the inner loop for the last two passes over the input data. The twiddle factors involved in the last two passes are either 1 or –j, which means that the complex multiplication is no longer necessary. Additionally it is possible to efficiently implement these last two passes in one inner loop. This optimization was included in both our scalar and Altivec versions of the FFT.

The last step of the FFT is to reorder the outputs according to a bit-reversed index order. Initially we reused the code that the Ptolemy VIS FFT [7] used to perform this task. However, we discovered that calculating the bit-reversed values at run-time was costly; approximately 50% of the FFT cycles for our Altivec implementation were spent reordering the output buffer. For this reason, we improved our Altivec star to pre-calculate the bit-reversed ordering, so that the only work necessary at runtime for a 256 point FFT is a sequence of 120 swap operations.

4 Simulation Results

4.1 FIR Kernel Results

Table 1 shows the results for the FIR kernel tests and the Ptolemy IIR demo. The IIR demo uses two FIR actors to represent an IIR filter.

Table 1 FIR Simulation Results

IMPLEMENTATION	CYCLE COUNT	INSTR. COUNT	CODE SIZE
8-tap Scalar	61,464	52,800	23,496
8-tap Motorola Altivec*	47,522	43,211	24,408
Our 8-tap Altivec	47,065	41,939	24,024
16-tap Scalar	67,068	68,800	23,528
Our 16-tap Altivec	50,083	45,811	24,472
IIR demo (16-tap) Scalar	96,245	105,953	23,736
IIR demo (16-tap) Altivec	60,569	56,063	25,192

The reason we do not see an 8 times increase in performance, as was theorized in Section 3.2, is because the number of instructions used for the packing and unpacking of the data is on the same scale as the number of instructions used in the FIR kernel itself. The actual speedup of the 16-tap FIR kernel is shown to be 1.34 over the scalar implementation. The speedup for the 16-tap IIR implementation is 1.59 over the scalar version. The reason the IIR speedup is higher than the single FIR kernel test speedup is that the IIR utilizes two FIR which means that the packing and unpacking sections of the code are a much smaller part of the overall code. Ptolemy provided a visual representation of this partition between the scalar sections and the vectorized sections of the code.

4.2 FFT Kernel Results

We measured our FFT implementations to have a 39.94 decibel signal to noise ratio using a floating point FFT as a reference with a complex exponential input signal. Performance results for the FFT kernel operating in a minimal system similar to the VIS FFT demo (as in Ptolemy 0.7.1 package) are shown in Table 2. The first and second rows correspond to processing a single

256 point FFT (with characteristics as described in Section 3), neglecting overhead due to packing and unpacking scalar values to and from vectors for the Altivec version. In the first row, the bit-reversed indices for the reordering of the FFT output are calculated at run-time, while they are computed at compile-time in the second row.

The speedup is noticeably improved when less clock cycles are used to reorder the outputs, since this is a component of the cycles for both versions of the FFT. The Altivec speedup for the system is considerably lower than when only the FFT is considered. The common overhead of generating the source signal and saving it to a buffer is one cause for the decreased speedup in an analogous fashion to the effect of a slower reordering algorithm being used in both versions. The vector/scalar data format conversion also causes a portion of the cycle reduction to be negated, lowering the speedup in the third row. However, a trace of the pack and unpack stars shows that if the memory accesses for these stars hit in the cache, then the entire overhead for vector-scalar conversion is approximately 3,200 cycles. This indicates that the system overhead is the main cause for speedup reduction in the minimal demo system.

The Sun VIS implementation of a 256 point FFT was compared to a floating point FFT implementation in [7] with a speedup of 1.28. Although it is unclear what scalar FFT implementation was used for comparison, it should be noted that the FFT star distributed with Ptolemy's CGC domain does not include performance optimizations (such as pre-computing twiddle factors) that are included in the VIS star implementation. Regardless, he performance enhancement of Altivec over the scalar FFT version certainly compares favorably with the speedup obtained using Sun's VIS extensions.

Table 2: FFT Simulation Results

CONFIGURATION	SCALAR CYCLES	ALTIVEC CYCLES	ALTIVEC SPEEDUP	SCALAR INST COUNT	ALTIVEC INST COUNT
FFT Only (basic reorder)	85,672	16,363	5.24	74,915	13,126
FFT Only (enhanced reorder)	82,431	11,654	7.07	68,573	7,063
Minimal demo (10 iterations)	1,814,663	1,230,379	1.47	1,651,504	1,074,377

4.3 Demonstration System Results

Table 3 shows performance results for ten iterations of our fixFIR demo. Our fixFIR demo system is organized the same as it is in the Ptolemy distribution with the exception that our scalar and Altivec versions of the FFT and FIR stars are used in place of the standard Ptolemy CGC stars. The key components of the demonstration are three FIR filters that feed data to three FFT filters. We believe the speedup numbers for the fixFIR demo are superior to the minimal system demonstrations for the FFT and FIR for two reasons. Firstly, the overhead of vector to scalar conversion is lower since the FIR passes data in vector format to the FFT galaxy, effectively amortizing the cost of the conversion. Secondly, a greater majority of the computation time in this system is spent in the kernels since the ratio of kernel actors to source actors is higher than in the minimal demo systems. It is interesting to note that the Altivec version was superior in terms of code size, dynamic instruction count, and cycle count for this demonstration. Selection of scheduler did not play a big role in the performance results (in fact the SJS scheduler produced a schedule identical to the cluster scheduler) probably due to the minor complexity of the graph.

Table 3 fixFIR Simulation Results

SCHEDULER	SCALAR CYCLES (1000S)	ALTIVEC CYCLES (1000S)	ALTIVEC SPEEDUP	SCALAR INST. CNT (1000'S)	ALTIVEC INST. CNT (1000'S)	SCALAR CODE SIZE	ALTIVEC CODE SIZE
ACY	4,997.5	1,514.6	3.30	5,074.0	1,605.6	43,542	39,244
Cluster	4,978.0	1,644.1	3.03	5,071.1	1,681.5	43,500	39,404

5 Conclusion

In summary, the scheduling and partitioning of the vectorized and scalar sections of a program are paramount in writing effective Altivec code; The Ptolemy environment facilitates the examination of these two aspects of programming by (1) allowing the flexibility to chose certain models of computation that ultimately effect the scheduling; and (2) providing a graphical representation of the vector – scalar partition. Ptolemy's strengths as a tool for partitioning systems have been described in previous literature. The authors have shown that these strengths can be extended to include software vector-scalar partitioning for fast functional design using NSP with Altivec in the Ptolemy environment

References

1. P. Ranganathan, S. Adve, and N.P. Jouppi, "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions," *Proc. ACM/IEEE Int. Sym. on Computer Architecture*, May 1999, pp.124-135

2. D. Bearden, D. Caffo *et al*., "A 780 MHz PowerPC Microprocessor with Integrated L2 Cache", *ISSCC* 2000

3. J. Tyler, J. Lent, A. Mather, and H. Nguyen, "Altivec: Bringing Vector Technology to the PowerPC Processor Family," *1999 IEEE International Performance, Computing and Communications Conference*, Feb. 1999, pp. 437-44.

4. *Altivec Programming Environments Manual*, Motorola Inc., 1999.

5. *Altivec Programming Interface Manual*, Motorola, Inc., 1999.

6. T.M. Conte, P.K. Dubey, M.D. Jennings, R.B. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, and A. Wolfe, "Challenges to Combining General-Purpose and Multimedia Processors," *IEEE Computer*, Dec. 1997, vol.30, no.12 p.33-7.

7. W. Chen, H.J. Reekie, S. Bhave, E.A. Lee, and A. Singh, "Native Signal Processing on the Ultrasparc in the Ptolemy Environment," *Proc. IEEE Asilomar Conference on Signals, Systems and Computers*, Nov. 1996, vol. 2, pp. 1368-72.

8. R. Bhargava, L.K. John, B.L. Evans, and R. Radhakrishnan, "Evaluating MMX Technology using DSP and Multimedia Applications," *Proc. ACM/IEEE Int. Sym. on Microarchitecture*, Nov. 1998, pp.37-46.

9. A. Kalavade and E. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," *IEEE Design and Test of Computers*, Sep 1993, vol. 103, pp. 16-28.

10. Motorola, "The Altivec Information Source", http://www.altivec.org

11. Apple, "Apple Developer Connection Download Page", http://developer.apple.com/hardware/altivec/download_summary.html

12. Daubechies, I., "Orthonormal Bases of Compactly Supported Wavelets", *Comm. Pure and Applied Math.*, vol. 41, Nov. 1988, pp. 909-996.