AltiVec Extensions to the Portable Expression Template Engine (PETE)*

Edward Rutledge
MIT Lincoln Laboratory

Abstract

Two simultaneous and conflicting goals of a high performance embedded signal processing library are expressiveness and efficiency. As library designers, we seek to provide an interface that allows signal processing algorithms to be expressed in a mathematical form, which simplifies the process of translating from algorithm specification to implementation. The Portable Expression Template Engine (PETE) [1] helps us to create such a high-level interface for a C++ signal processing library, without sacrificing efficiency. In this work, we examine the suitability of PETE for the Motorolla G4 architecture, with its SIMD AltiVec extensions. Our experiments show that PETE is suitable for use on the G4 architecture.

Introduction

By supporting efficient, high-level, composable C++ operator and function implementations of element-wise multi-dimensional array operations, the Portable Expression Template Engine (PETE) assists an embedded C++ signal processing library in achieving the goals of expressiveness and efficiency. For instance, a library utilizing PETE supports expressions like A=B+C-D, where A, B, C and D are vectors, and where the processor efficiency for executing the statement is similar to that of a hand coded C/C++ loop. However, PETE typically is used to operate on one element of the array(s) at a time, which is inefficient when SIMD instructions are available, as is the case with the G4 processor architecture.

The Motorolla G4 processor architecture has gained popularity as an embedded signal processing platform due to its potential for favorable power to performance ratio. In order to realize this favorable power to performance ratio, a programmer must take advantage of the G4's SIMD AltiVec unit, which enables operations on 128 bit values, allowing operations across multiple array elements (for example 4 floating point values) at once. C/C++ language extensions have been defined to allow programmers to take advantage of the G4's AltiVec unit, and are now supported by many compilers. In this work, we show how to use the AltiVec C/C++ extensions along with PETE to achieve both a high level of performance and a high level of abstraction in a signal processing library running on the G4. Our experiments show that an experimental vector class using PETE performs about as well as a hand-coded loop that makes use of the AltiVec extensions, performs slightly worse than an implementation using an optimized VSIPL library (Vector Signal and Image Processing Library) [3] for expressions consisting of only one operation, and performs better than an implementation using an optimized VSIPL library for expressions consisting of more than one operation or expressions operating on small data objects. This performance difference is due to the nature of the VSIPL C language binding, not to deficiencies in the particular implementation of VSIPL; because VSIPL cannot take advantage of the C++ expression template [2] and inlining features, VSIPL cannot combine several element-wise operations to produce an implementation with performance equivalent to a hand coded *for* loop. PETE, on the other hand, which was built to take advantage of those features of the C++ language, can. In addition to having good performance characteristics, the PETE implementation results in more straightforward application code than either the hand coded loop or VSIPL implementations.

Experiments

To show the performance characteristics of PETE using AltiVec, we compare it to two other implementations: a hand-coded C/C++ loop implementation using the AltiVec extensions, and an implementation using an optimized implementation of the VSIPL core-lite profile. We implement a variety of expressions involving vectors of floating point values, ranging from simple one-operation expressions, up to expressions consisting of 4 or more operations. We include multiply-add operations in our experiments, which are of particular relevance to signal and image processing, and which are directly supported by the AltiVec architecture in the form of a 1 cycle multiply-and-accumulate instruction.

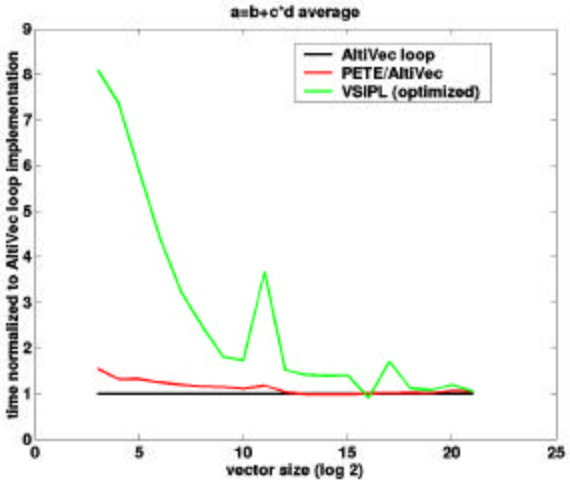
We run our experiments on a single Power PC 7400 (G4) processor with a 450 MHz clock residing in a DY4 CHAMP-AV quad G4 processor board and running the VxWorks operating system. The processor has 64KB of L1 cache, which is split into 32KB each for data and instructions, 2MB of L2 cache, and access to 128MB of local SDRAM, of which it is configured to use 64 MB. To compile our tests, we use a GCC 2.95.4 Solaris to G4/VxWorks cross-compiler, which is an unofficial release based on GCC 2.95.3, but with support for the C/C++ AltiVec extensions, and some patches to improve VxWorks compatibility. We use a VSIPL core-lite implementation, purchased from MPI Software Technology

* This work is sponsored by the US Navy, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the Department of Defense.

(website: www.mpi-softtech.com), specifically optimized for the G4/VxWorks target.

Results

This section contains a subset of the results we will present. In it, we show experimental results for the expression *A=B+C*D*, where *A*, *B*, *C*, and *D* are vectors of length *N*. Figure 1 shows the performance of all 3 implementations (hand coded AltiVec loop, PETE with AltiVec, optimized VSIPL).



The horizontal axis is $log_2 N$. The vertical axis is the time required by each implementation to perform the operation, normalized to the time of the hand coded AltiVec loop implementation, averaged over many iterations.

There are several interesting features in this figure. Note that the hand coded loop and PETE implementations outperform the optimized VSIPL core-lite implementation of this particular expression by at least 40% for vectors of size 2^{15} (32K) elements and less. There are two reasons for this. First, the VSIPL implementation makes function calls to the VSIPL library, which cannot be inlined by the compiler because the VSIPL library functions are pre-compiled object code. The hand coded loop implementation of course does not contain anything that needs to be inlined, and the compiler does inline the PETE implementation, so its performance is nearly equivalent to that of the loop. Second, because we are using the core-lite profile of VSIPL, there is no vector multiply-add function available. Instead, in the VSIPL implementation of the expression, the multiply of C and D must be performed, the results stored in a temporary vector, and then the temporary vector added to B. This results in reduced performance both because the AltiVec multiply-add instruction cannot be used, and because the materialization of the intermediate temporary vector results in poor cache usage. If the VSIPL multiply-add function were available to us (as it is in the core profile of VSIPL), we would probably see the performance of the VSIPL implementation become roughly equivalent to that of the other two implementations at vector sizes around 256 or 512 elements (which is the behavior observed for a simple vector add, the results of which will be shown in the full presentation).

Also note the "spikes" in the VSIPL implementation's time compared to the other two implementations at vector sizes of 2^{11} (2K) and 2^{17} (128K). These are once again due to the necessity of storing the results of the VSIPL multiply in a temporary vector before performing the add. These spikes are the result of the VSIPL implementation's overflowing the L1 cache and L2 cache, respectively, before the other two implementations do because of the extra temporary vector required to hold the results of the multiply. Again, if the VSIPL multiply-add function were available to us, these spikes would not appear.

Finally, note that the VSIPL implementation outperforms the other two by a slight margin at vector length 2^{16} and that execution time for the 3 implementations is roughly equivalent once they overflow L2 cache. Similar results are seen in implementations of other expressions. Further analysis and investigation is required to determine why this is the case.

Conclusions

We have shown that PETE is compatible with the G4 processor architecture and its SIMD AltiVec extensions. We have demonstrated how to incorporate the AltiVec C/C++ extensions into element-wise operators for a vector class implemented using PETE. We have also shown that these operators perform nearly as well as a hand-coded loop, and outperform an optimized VSIPL implementation for small vector sizes or non-trivial expressions. The difference in performance is due to the nature of the VSIPL C binding, not to deficiencies in the VSIPL implementation. In addition to the performance benefits, the PETE element-wise operators with AltiVec extensions support a higher level of abstraction, easing the transition from a mathematical algorithm specification to its high performance implementation.

References

[1] Scott Haney, James Crotinger, Steve Karmesin, and Stephen Smith. PETE, the Portable Expression Template Engine. *Dr. Dobbs Journal*, 1999

[2] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26-31, 1995.

[3] David A. Schwartz, Randall R. Judd, William J. Harrod, and Dwight P. Manly. *VSIPL 1.02 API*, 2002.

* This work is sponsored by the US Navy, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the Department of Defense.