Monolithic Compiler Experiments using C++ Expression Templates*

Lenore R. Mullin, Edward Rutledge, Robert Bond
MIT Lincoln Laboratory**

Abstract

In this work, we study and prototype compiler optimizations for array indexing operations. The optimizations can be proven theoretically correct by the Psi Calculus, developed by Mullin [5], and result in very efficient implementations in which the need to materialize intermediate array values in complex expressions is eliminated. Our prototype implementation is based on C++ expression templates [8], and the Portable Expression Template Engine (PETE) [1] developed at the Advanced Computing Laboratory at Los Alamos National Laboratory. Thus, it consists of a set of C++ class and function templates that can be included and used by a C++ program or class library. We test the performance of our optimizations by applying them to compositions of 1-D indexing operations. We then compare the performance of our optimizations to that of a hand coded C/C++ loop implementation, and a naïve C++ implementation where intermediate array values are materialized after each indexing operation. We show that our optimizations result in an implementation whose performance is nearly as good as a hand coded loop implementation, whose interface achieves the high-level of abstraction of a naïve C++ implementation, and whose performance far outstrips that of a naïve C++ implementation.

Introduction

Monolithic programs consist of statements that execute directly on multi-dimensional array objects that are considered basic types in a language. This level of expressiveness is desirable not only because a large class of signal processing and scientific computing applications consist of operations on arrays, but also because at the monolithic level, regularity of access patterns and array structure can be abstracted and made visible to a language or compiler underneath, assisting in transforming the statements into efficient implementations. An implementation that can take advantage of such regularity can achieve performance rivaling that of hand crafted code. With decomposition and mapping strategies formalized and abstracted to the same high level, optimization over diverse memory hierarchies and parallel architectures can be algebraically derived. When followed by the scalarization techniques employed in modern compilers, the result is highly optimized and efficient code that can be theoretically proven to have minimal memory accesses due to the materialization of unnecessary intermediate values and temporary variables.

When rules from the Psi calculus are applied, any expression of array index manipulation operations, such as circular shift, reverse, concatenate, and transpose, can be reduced to a series of loops, where each loop assigns part of an operand to part of the result. Consider *A=reverse(B++C)*, where ++ is the array concatenation operator, reverse simply reverses the vector, and *A*, *B*, and *C* are 1-D arrays. The expression can be converted into 2 loops, the first assigning from *B* to the end of *A*, and the second assigning from *C* to the beginning of *A*. This is the most efficient implementation of the expression in terms of the number of memory reads and writes. In fact, we can prove that, for any given array expression, reduction rules from the Psi Calculus can be applied in a mechanical process guaranteed to produce an implementation having the least possible number of memory reads and writes [5]. Psi reduction has been prototyped as a compiler for an MOA (Mathematics of Arrays) language [4]; a compiler preprocessor for Fortran [6]; a compiler for Single Assignment C (SAC), which is an extended version of C [3]; and a set of C++ classes [2].

To explore similar compiler-style optimizations to C++, we used expression templates. We augmented the Portable Expression Template Engine (PETE) to implement this same reduction process, essentially providing a C++ to C++ "compiler" for converting array expressions to a series of *for* loops. This implementation allows a C++ high-performance embedded signal processing library to easily provide efficient, composable, high-level array operations.

Experiments

To show the performance characteristics of our expression template implementation of array operations, we compare it to two other implementations: a hand-coded C/C++ loop implementation, and a naïve C++ implementation in which each array operator returns an intermediate array result. The particular algorithm we have chosen to implement is a time-domain convolution of two 1-D vectors. We chose this algorithm both because it is common in signal processing, and because it requires a combination of several basic array operations, such as take, drop, and concatenate, and arithmetic operators such as sum, for its implementation. This latter quality is important in allowing us to demonstrate the composability of array operations in our approach. That is, we

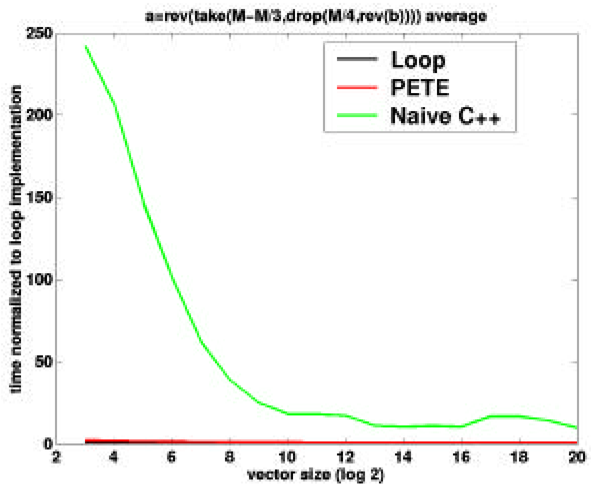
* This work is sponsored by the Defense Advanced Research Projects Agency, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the Department of Defense.

** Dr. Mullin participated in this work while on sabbatical leave from the Dept. of Computer Science, University of Albany, State University of New York, Albany, NY can combine low-level array operations to easily build efficient high-level operations. The choice of 1-D array objects was for convenience; [5] shows how the Psi calculus concepts being demonstrated apply to multi-dimensional arrays.

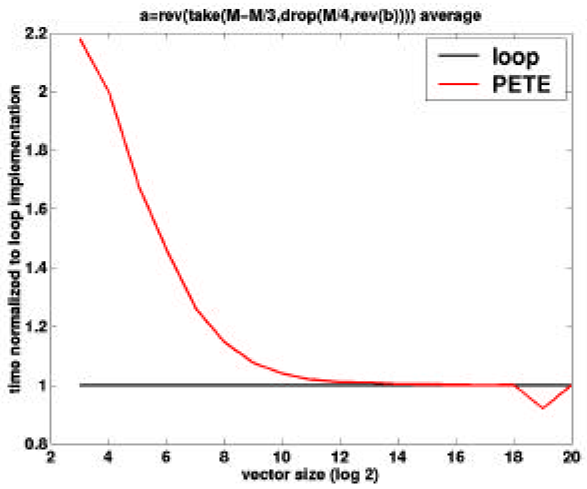
We run our experiments on a single Power PC 7400 (G4) processor with a 450 MHz clock residing in a DY4 CHAMP-AV quad G4 processor board and running the VxWorks operating system. The processor has 64KB of L1 cache, which is split into 32KB each for data and instructions, 2MB of L2 cache, and access to 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.

Results

Although we do not have experimental results for convolution as of the time of this writing, we will be presenting those results in our full presentation. We do at this time have experimental results for several low-level 1-D index manipulation operations, including take, drop, and reverse. *Reverse* simply reverses a 1-D array. *Take(N, A)*, where *A* is a 1-D array of length *M*, and *N* is a non-negative integer, results in a 1-D array of length *N*, whose contents are the first *N* elements of *A*. *Drop(N, A)* under the same assumptions results in a 1-D array of length *M-N*, whose contents are the last *M-N* elements of *A*. *Take* and *drop* are both used in composing the convolution operation. We present experimental results here for the expression:
A=reverse(take(M-M/3,drop(M/4, reverse(B)))), where *B* is length *M*. Figure 1 shows the performance of all 3 implementations (hand coded loop, enhanced PETE, naïve C++).



The horizontal axis is $log_2 M$. The vertical axis is the average time of each implementation to perform the operation, normalized to the average time of the hand coded loop implementation. Note that the loop and expression template implementations significantly outperform the naïve C++ approach, with the latter taking roughly 200 times as long as the loop implementation for small (M<16) arrays. Figure 2 compares only the performance of the hand coded loop and enhanced PETE implementations.



Note that the enhanced PETE implementation incurs some overhead, which is apparent for small vector sizes (for this expression, the expression template approach takes over twice as long as the hand coded loop when *M*=8). This overhead is mostly due to the fact that the expression must go through several transformation steps prior to processing, and since the arrays are dynamically sized, some of these steps must be performed at run time. However, there are ways to mitigate this overhead. We will discuss and present results obtained from 2 of these: a setup step to be performed at system startup time for each subsequent assignment in the system, and static (compile time) sizing of array variables. Each approach has advantages and disadvantages, but both should largely eliminate the remaining overhead in the enhanced PETE implementation of the index manipulation operations compared to the hand-coded loop implementation.

Conclusions and Future Work

As is the case with element-wise algebraic array operations, expression templates can be used to implement array index manipulation operations with an efficiency that rivals that of hand coded loop implementations, while providing the high level of abstraction of naïve C++ implementations. We have demonstrated how to achieve this by augmenting PETE to implement Mullin's Psi Calculus reduction rules for a 1-D array class. In this abstract, we have presented results demonstrating that such an implementation performs nearly as well as a hand coded loop implementation, and far outperforms a naïve C++ implementation, for low-level index manipulation operations. In our full presentation, we will show results for a convolution of two 1-D vectors, which should confirm our initial results. Future work will include extending this work to multi-dimensional distributed arrays, and extending this work beyond single expressions to implement broader program block optimizations, as described in [7].

References

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

[2] Manal Helal and Ahmed Smeh. Dimension and Shape Invariant Programming: The Implementation and the Application. In *Proceedings of 3rd WSEAS Symposium on Mathematical Methods and Computational Techniques in Electrical Engineering*, December 2001.

[3] L. Mullin, W. Kluge, and S. Scholtz. On Programming Scientific Applications in SAC – a Functional Language Extended by a Subsystem for High Level Array Operations. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages*, Bonn, Germany, 1996.

[4] L. Mullin and S. Thibault. Reduction Semantics for Array Expressions: The PSI Compiler. Technical Report CSC94-05, Dept. of CS, University of Missouri-Rolla, 1994.

[5] L. M. R. Mullin. A Mathematics of Arrays. PhD thesis, Syracuse University, December 1998.

[6] L. R. Mullin, D. Eggleston, L. J. Woodrum, and W. Rennie. The PGI-PSI Project: Preprocessing Optimizations for Existing and New F90 Intrinsics in HPF Using Compositional Symmetric Indexing of the Psi Calculus. In *Parallel Computers*, pages 345-355, Aachen, Germany, December 1996. Forschungszentrum Julich GmbH.

[7] D. Rosenkrantz, L. Mullin, and H. B. Hunt III. On Materializations of Array-Valued Temporaries. *Lecture Notes in Computer Science*, 2017:127-141, 2002.

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