Generative Programming Considerations for the Matrix-Chain Product Problem

Andrew A. Anda
Computer Science Department
St. Cloud State University
St. Cloud, MN 56301
aanda@stcloudstate.edu

1 March 2008

Abstract

Many who utilize C++ matrix classes which implement overloaded arithmetic operators are unaware that by allowing the matrix-chain product expressions they code to be evaluated via the multiplicative operator's intrinsic associativity, the performance of the evaluation might be significantly suboptimal relative to the performance of the optimal ordering. We propose that such a matrix class should be augmented to automate the process of identifying matrix-chain products in source code, determining an efficient ordering, and evaluating the matrix-chain using that ordering. We discuss considerations concerning: the influence of special algorithms and matrix types, the completeness of static specifications, the implementation environment, and future language standards.

Andrew A. Anda
Computer Science Department
St. Cloud State University
St. Cloud, MN 56301
aanda@stcloudstate.edu

1 Introduction

1.1 Overview

When a specialist in any specific discipline, an application domain, needs to formulate and implement a computational solution to a problem in that domain, they will specify, analyze, formulate an algorithm, then encode that algorithm into some programming environment, translating syntax and semantics from their domain into that of the programming environment. The more correlated the syntax and semantics of the programming language is to the application domain, the more efficient and reliable that process should prove to be, and the less the domain specialist is required to know about how to efficiently and reliably perform that translation to the programming environment. One aspect of the evolution of high-level programming languages is the ongoing goal of further *deskilling* the programmer, by automating routine tasks, raising the abstraction level of the interface, establishing reasonable, common, and expected (overridable) defaults, and enhancing the insulation of the programmer from implementation details. Seminal early high-level programming languages targeted specific broad application domains, e.g. COBOL for business and FORTRAN for science and engineering. Soon thereafter, the goal of more specific domain expression was satisfied via the development of source libraries of related modular subprograms. Examples in the context of numerical linear algebra include the BLAS, EISPACK, and LINPACK. Some years later, the trend emerged to design high-level languages to be primarily general-purpose. For these languages, external facilities for domain-specific specialization are essential. The *class* facility in object-oriented languages representing an abstract data type has evolved from the preceding library paradigm as the means to represent specialized domain knowledge and operability.
C++ evolved from C as an alternative to SIMULA67.[19] First classes, then operator overloading and templates were introduced. Operator overloading furthers the goal of matching the syntax and semantics of a specific application domain to that of the programming language. The operator overloading facility in C++ is not as general as one might want, as the set of overloadable operators is restricted to a well defined subset of the set of intrinsic operators with no allowance for overriding the default precedence and associativity of and operator (e.g. overloading the caret '^' operator for exponentiation yields the wrong associativity). Templates, introduced to support generic programming, accidentally included an ability to perform static (compile-time) computations and even code generation.[25] Templates can contain recursive expressions. In fact, the template layer itself provides Turing-completeness. [22, pp. 312–313] [8, p. 407]

1.2 The Matrix Arithmetic Domain

Computational problems in an extensive set of disciplines are routinely solved via at least partial formulations as the solution of matrix algebra problems. A matrix facility in a programming language will better match its targeted domain if the arithmetic operators for addition, subtraction, and multiplication are overloaded for scalars, vectors, and matrices. Whereas matrices and their arithmetic operators are intrinsic in Fortran90 and its successors, in C++ these must be defined as extrinsic classes. Object oriented programming raised

the overall abstraction level, but at the expense of run-time performance.
Templates can provide type *genericity*, i.e. static specializations that allow the arithmetic operators to abstract commonality by processing different types of matrices, with minimal code duplication, uniformly or specially.[2] Additionally, templates can be exploited *generative*ly to statically handle special cases and conditionals, evaluate expressions, and optimize code for execution (e.g. temporary reduction or loop unrolling) – this is termed *metaprogramming*. [24, 23, 25, 18, 8, 7, 1] Early matrix class libraries exploited templates for their generic facility only. Subsequent and more contemporary matrix class libraries (e.g. Blitz++, POOMA, MTL, and GMCL) implement templates metaprogramming methodologies to optimize run-time performance on a variety of matrix types.[26, 18, 17, 6]
Matrices can multiply inherit a wealth of common properties including

precision e.g. single, double, quad

number algebra type e.g. integer, rational, real, complex

symmetries e.g. symmetric, non-symmetric, per-symmetric, Hermitian

density e.g. dense, sparse

patterned e.g. diagonal, banded, (upper/lower): triangular, Hessenberg

storage format e.g. CSR, CSC, recursive (tiling pattern: RBR, RBC, Hilbert, Z-Morton)

rank e.g. full rank, rank-one, rank-two, ...

structure e.g. circulant, Hankel, Vandermonde, Cauchy, Toeplitz, Fourier

special e.g. Hilbert, Krylov, stochastic

shape e.g. square, rectangular

spectral properties e.g. SPD

blocking

Without generics, a library handling each case would suffer from exponential combinatorial bloat. However, templates can be exploited to manage this complexity automatically. E.g. the GMCL library provides multiple template parameters for the matrix type and feature categories covering 1840 kinds of matrices with roughly 7500 lines of code.[26] Different matrix types and features are handled by a matrix configuration generator which is called by a matrix expression generator wherein overload operators construct expression objects representing the structure of the expression they are used in. The evaluation of the tree of expression objects is handled by the assignment operator.[8]
The overarching principal is that if a property is available at compile-time, it can be exploited at compile-time. Therefore, we can at compile-time perform partial, or even total, expression evaluation.

2 Motivation

If users learn about the properties of matrix algebra in a typical linear/matrix algebra course in the mathematics curriculum, they most likely learn that with respect to multiplication, matrices in general don't commute. But, matrices do associate. So, all associative orderings of the *matrix-chain* product of conformal matrices,

$$\prod_{i=1}^{n} A^{k_i \times k_{i+1}} \tag{1}$$

will give the same answer. However, math students are seldom introduced to the property that in general, for a matrix-chain product of conformal matrices, some of which are not square, associativity does not hold with respect to the total number of scalar operations. It is likely that a specialist, in a domain unrelated to computer science and combinatorics, would not know that by relying on the default left-to-right associativity of the multiplication operator, they might be performing an excessive and sub-optimal number of scalar products. For that reason, this instance of specialized matrix domain knowledge should be integrated into a generative matrix library. That goal is the focus of this paper.
The task of determining exhaustively the optimal (fewest operations) associative ordering becomes infeasible for longer product chains of matrices because the number of possible orderings grows proportional to the sequence of *Catalan numbers* which grow at the exponential rate of $\Omega(4^n/n^{3/2})$.[5, pp. 331–349] The optimal ordering can be determined in polynomial time $\Omega(n^3)$ and space $\Omega(n^2)$[16, 10, 5] by performing *dynamic programming* [3] or *memoization* [5, pp. 347–349]. Hu and Shing developed lower complexity optimal solution algorithm with $O(n \log n)$ $O(n)$ space requirements.[14, 15] Other lower complexity optimal matrix-chain algorithms have been developed which require either parallel processing, or finding a sufficiently optimal approximate solution.[4, 9]

3 Proposal

The preceding section motivates a proposal for research and development towards the following objective: create an C++ facility which

1. identifies matrix-chain products in source code,
2. determines an efficient ordering,
3. evaluates the matrix-chain using that ordering,

via C++ template metaprogramming.

3.1 Considerations

3.1.1 Static/Dynamic

The proportion of work that can be performed at compile-time will be dependent on the degree of completeness of the set of parameterized features at compile-time. If any of the

matrices are dynamic or if any array slice dimensions are only known at run-time, then the class can perform the full ordering determination only at run-time. However, fully static subchains can be processed statically for optimality as any optimization problem amenable for solution via dynamic programming exhibits *optimal substructure*, i.e. an optimal solution comprises optimal subproblems.[5, p. 339]
The class will establish a default algorithm for the ordering determination. As it is likely that most matrix-chain products will be short, the simplest efficient algorithm should suffice. More efficient ordering algorithms, if they are part of the class, could either be selected through declaration by the user, or could be selected conditionally based on chain length. Additionally, a user could select an approximating algorithm. A facility whereby the user may optionally provide their own algorithm via functor would add desirable extensibility.

3.1.2 Binary product operation counting

Current examples of implementation of the matrix-chain product assume that the standard cubic complexity binary matrix product is applied to a fully dense and general rectangular matrix. However, if more specific characterizations of any of the two operands of a binary matrix product such as those itemized in the far-from-exhaustive list [1.2] of matrix types, those characteristics can be exploited by a more efficient algorithm yielding lower operation counts and/or execution rates than the standard general product algorithm. Even general matrices can be multiplied more efficiently, and with a lower computational complexity,(if square) via one or more recursive applications of the Strassen-Winnograd matrix product algorithm.[11, 13] Similarly, the more efficient 3M algorithm can be used for general complex matrix products.[13, pp. 437–438] If accurate operation counts for the products of special matrices and algorithms are provided to the ordering optimizer, then the optimal ordering may change. Therefore there should be a facility which provides a reliable exact or approximate operation count for any pair of matrices and product algorithm on those matrices. The operation counts could be generated *a priori* or alternatively, benchmarking can be used to generate functions which estimate runtime performance. Because of the significance of locality on modern computer architectures for algorithm performance, the relationship between operation count and performance is not as tightly coupled as it was pre-NUMA.

3.1.3 Implementation environment

Rather than start from scratch, this project would be more feasibly performed by augmenting an existing metaprogrammed C++ matrix library. Although, a stand-alone class could be feasible and practical if it is restricted to ordinary dense general matrices and the standard product algorithm. This might be a good target for a proof-of-concept project. Because the current template facility is Turing-complete, it is certainly powerful enough to perform everything we propose. However, certain essential generator and type related facilities have to be performed rather awkwardly and opaquely in the current C++ standard. The next major revision of C++ [C++0X] will add critical syntax and semantics (e.g. *concepts*) that will permit a more elegant implementation of generic and generative metaprogramming techniques.[12]

3.1.4 Beyond matrix-chain products

A more general project would be to parse matrix expressions to extract other optimizable sub-expression types. E.g. matrix polynomials of the types

$$\sum_{i=1}^{n} a_i X^i \tag{2}$$

$$\sum_{i=1}^{n} A_i x^i \tag{3}$$

$$\sum_{i=1}^{n} A_i X^i \tag{4}$$

can be evaluated efficiently via Horner's method in conjunction with the binary power algorithm. However, equation [2] can be evaluated more efficiently than by Horner's.[13, pp.102–103] [16, 21] In fact, any function of a matrix can be extracted and evaluated with some being identified as having more efficient solution algorithms known. [11, Chap. 11] Additionally, existing generative matrix arithmetic and algebra libraries can be augmented and extended to include any of the almost inexhaustible supply of not-yet-handled matrix types for which efficient product algorithms are known.

4 Summary

Because many who use matrix classes (having overloaded arithmetic operators) are unaware of the criticality of the associative ordering of the matrix-chain product expressions they code, we propose that the class should be augmented to automatically identify and evaluate the matrix-chain product using the optimal or a close-to-optimal ordering. This objective can be implemented via C++ template metaprogramming. This project can be partitioned into a set of coupled sub-projects. Extensions of this project to specific matrix types and expression types could spawn a wealth of related projects.

References

[1] ALEXANDRESCU, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley, Boston, MA, USA, 2001.

[2] BARTON, J. J., AND NACKMAN, L. R. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison Wesley, Reading, MA, USA, 1994.

[3] BELLMAN, R. E. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, USA, 1957.

[4] CHIN, F. Y. An o(n) algorithm for determining a near-optimal computation order of matrix chain products. *Commun. ACM 21*, 7 (1978), 544–549.

[5] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, second ed. The MIT Press, Cambridge, MA, USA, 2001.

[6] CZARNECKI, K. *Generative Matrix Computation Library home page*. 2006, `http://nero.prakinf.tu-ilmenau.de/~czarn/gmcl`.

[7] CZARNECKI, K., EISENECKER, U., GLÜCK, R., VANDEVOORDE, D., AND VELDHUIZEN, T. Generative programming and active libraries (extended abstract). In *Generic Programming. Proceedings* (2000), M. Jazayeri, D. Musser, and R. Loos, Eds., vol. 1766 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 25–39.

[8] CZARNECKI, K., AND EISENECKER, U. W. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, Boston, MA, USA, 2000.

[9] CZUMAJ, A. Very fast approximation of the matrix chain product problem. *J. Algorithms 21*, 1 (1996), 71–79.

[10] GODBOLE, S. S. On efficient computation of matrix chain products. *IEEE Transactions on Computers C-22*, 9 (1973), 864–866.

[11] GOLUB, G. H., AND VAN LOAN, C. F. *Matrix Computations*, third ed. The Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[12] GREGOR, D., JÄRVI, J., SIEK, J., STROUSTRUP, B., REIS, G. D., AND LUMSDAINE, A. Concepts: linguistic support for generic programming in c++. *SIGPLAN Not. 41*, 10 (2006), 291–310.

[13] HIGHAM, N. J. *Accuracy and Stability of Numerical Algorithms*, second ed. the Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2002.

[14] HU, T. C., AND SHING, M. T. Computatation of matrix chain products. part i. *SIAM Journal on Computing 11*, 2 (1982), 362–373.

[15] HU, T. C., AND SHING, M. T. Computatation of matrix chain products. part ii. *SIAM Journal on Computing 13*, 2 (1984), 228–251.

[16] PATTERSON, M. S., AND STOCKMEYER, L. J. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing 2*, 1 (1973), 60–66.

[17] SIEK, J., AND LUMSDAINE, A. Software engineering for peak performance. *C++ Report* (May 2000), 23–27.

[18] SIEK, J. G., AND LUMSDAINE, A. The matrix template library: Generic components for high-performance scientific computing. *Computing in Science and Engineering 1*, 6 (Nov/Dec 1999), 70–78.

[19] STROUSTRUP, B. *The C++ Programming Language*, third ed. Addison Wesley, Reading, MA, USA, 1997.

[20] UEBERHUBER, C. W. *Numerical Computaton 1: Methods, Software, and Analysis*. Springer-Verlag, New York, NY, USA, 1997.

[21] VAN LOAN, C. F. A note on the evaluation of matrix polynomials. *IEEE Trans. Automat. Control AC-24*, 2 (1979), 320–321.

[22] VANDEVOORDE, D., AND JOSUTTIS, N. M. *C++ Templates: The Complete Guide*. Addison Wesley, Boston, MA, USA, 2003.

[23] VELDHUIZEN, T. L. Expression templates. *C++ Report 7*, 5 (1995), 26–31.

[24] VELDHUIZEN, T. L. Using c++ template metaprograms. *C++ Report 7*, 4 (1995), 36–43.

[25] VELDHUIZEN, T. L. C++ templates as partial evaluation. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999.* (Jan. 1999), University of Aarhus, Dept. of Computer Science, pp. 13–18.

[26] VELDHUIZEN, T. L., AND GANNON, D. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)* (Yorktown Heights, New York, 1998), SIAM Press.