IMPLEMENTATION OF AN OPTIMAL MATRIX-CHAIN PRODUCT EXPLOITING MATRIX SPECIALIZATION

Swetha Kukkala and Andrew A. Anda
Computer Science Department
St Cloud State University
St Cloud, MN 56301
kusw0601@stcloudstate.edu
aanda@stcloudstate.edu

Abstract

We discuss our implementation and development of a matrix-chain product evaluation facility in C++. The matrices in the product chain can be any of a set of predefined types. Our implementation consists of several basic components. The first component generates the essential matrix-chain operand characteristics. The second receives those characteristics then computes an optimal associative ordering using a memoized dynamic programming algorithm. The product can then be performed in that optimal associative ordering. In our implementation, we are extending the algorithm to be able to incorporate the scalar multiplication counts for various combinations of different matrix classes.

1 Introduction

A matrix product exists only if the pair of matrix operands are *conformal*, i.e. the number of columns of the left operand matrix must equal the number of rows of the right operand matrix, e.g. for the matrix product,$A^{i\times j}\,B^{k\times l}$, $j\ \equiv k$. The product of a sequence of conformal matrices is a *matrix-chain*, represented as:

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

The set of matrices is associative under multiplication. For example, the product of the conformal matrices *A*, *B*, & *C* yields the same value regardless of ordering, i.e.

$$(A \times B) \times C \equiv A \times (B \times C).$$

However, when the product of a sequence of conformal matrices, of differing sizes, is performed, the number of scalar multiplication operations required can vary depending on the associative ordering. The number of scalar multiplication operations is a direct function of the associative ordering of that product. For any matrix-chain, there will be one or more orderings that require the fewest scalar multiplication operations. We term those associative orderings requiring the fewest scalar multiplication operations *optimal* orderings. The size of the search space as a function of the number of matrices for the optimal ordering exhibits (exponential) combinatorial growth proportional to the growth of the sequence of Catalan numbers $\Omega(4^n/n^{3/2})$[1]. A greedy approach to finding an optimal ordering may fail with this type of problem, but an optimal solution will have the property of *optimal substructure* wherein the optimal solution is composed of optimal subproblems [2, p.327]. So we employ *dynamic programming* via *memoization* (maintaining a table of subproblem solutions) [2, p. 347] to compute an optimal ordering in polynomial time $\Omega(n^3)$ and space$\Omega(n^2)$. The algorithm we used is not novel. In fact, the matrix-chain optimality problem often motivates the pedagogical development and justification of the dynamic programming technique in standard algorithm analysis textbooks [2, p. 331; 9, p. 105; 8, p. 320; 7, p. 457].

Current examples of the matrix-chain product algorithm consider only operands which are only general matrices of the same precision [2]. But in practice, we can also consider specialized matrix types because the matrices can be sub-typed into a myriad of conventional categories, many of which have either differing operation counts for matrix products having the same complexity, or differing complexities for the product of the respective matrix operands [1]. Therefore, we provide a facility by which a class returns an optimal ordering of the matrix-chain product optionally with the number of operation counts.

Our objective is to provide an application programmer with classes which perform an optimal matrix-chain product using a minimum number of scalar multiplication operations. At this early stage, we are providing a proof of concept of the utility of exploiting matrix specialization. The matrix classes that we implemented augment those in the Template Numerical Toolkit (TNT) library. The TNT is an open source collection of interfaces and reference implementations of vector and matrix based numerical classes useful for scientific computing in C++ [4]. Our facility evaluates and extracts the matrix-chain operands coded by the programmer efficiently and returns an associative ordering of the matrices along with its scalar multiplication operation count (the measure we are minimizing).

1.1 Computation of scalar multiplication operation count

The product $C^{i\times j} = A^{i\times k} B^{k\times j}$, where A and B are general matrices, results in a general matrix, C, and requires the following number of scalar multiplication operations in equation

$$i \bullet j \bullet k \tag{1}$$

The general matrix-matrix product is an example of a *level* 3 BLAS (Basic Linear Algebra Subroutines) [10] operation. Changing the matrix type can change the scalar operation count. For example, we can pre- or post-multiply the dense matrix by a diagonal matrix.

A *diagonal* matrix has non-zero entries only on the main diagonal. Since the elements in the matrix off diagonal are all zero, there is no need to perform the scalar multiplication and additions involving those zero entries. Therefore, those operations will not be counted. Consider the following dense-diagonal matrix product,

$$\begin{pmatrix} 1 & 2 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1*1 & 2*1 & 4*1 \\ 4*1 & 5*1 & 6*1 \\ 7*1 & 8*1 & 9*1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{2}$$

We see in Equation (2) that the additions and multiplications with zero elements can be ignored. So the total number of scalar operations are 9 which is half less than the total number of operations when we apply (1). Therefore, multiplying an $i \times j$ dense matrix type with a $j \times j$ diagonal matrix type results in an $i \times j$ dense matrix. And the number of scalar multiplication operations is calculated using the equation

$$i \bullet j \tag{3}$$

A *triangular* matrix is a square matrix where the entries either below or above the diagonal are zero. Consider the following lower triangular-dense matrix product,

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1*1 & 1*2 & 1*3 \\ 2*1+3*4 & 2*2+3*5 & 2*3+3*6 \\ 4*1+5*4+6*7 & 4*2+5*5+6*8 & 4*3+5*6+6*9 \end{pmatrix} \tag{4}$$

We see in Equation (4) that the total number of scalar operations is 18 which is less than the total number of operations when we apply Equation (1), 27. Therefore, multiplying an $i \times j$ dense matrix with a $j \times j$ lower triangular matrix. The number of scalar multiplication operations is calculated using the equation

$$\sum_{a=1}^{j} a \bullet j \tag{5}$$

Our intention is to support the set of matrix types supported by the dense BLAS. Dense BLAS has become a *defacto* computation kernel of many libraries that perform matrix computations. We have not resolved how to design non squared patterned matrices.

1.2 Object Oriented Design

C++ supports *object oriented* programming which focuses on encapsulation, polymorphism, modularity and inheritance. Our matrix classes are based on those that are in the TNT following the principles of object oriented design. The specialized matrix classes that we developed are inherited from a basic matrix class in the TNT. We reused and extended the existing base class matrix via *virtualization*. Thus we achieved inheritance by obtaining the hierarchical relationships between the classes as shown in figure 1.

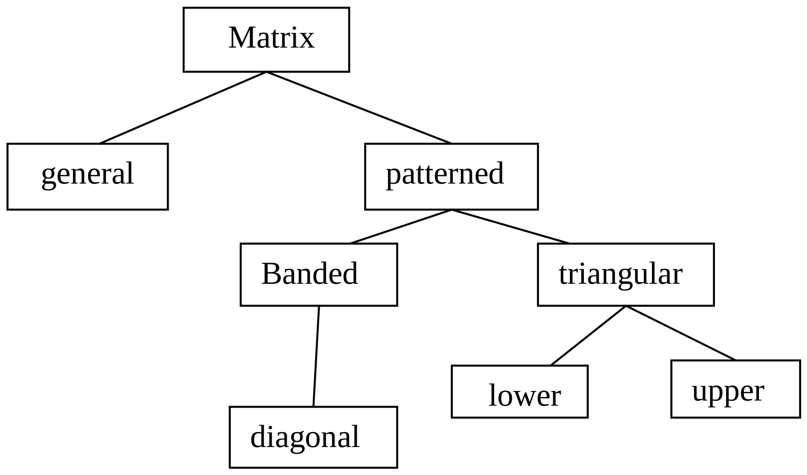

Figure 1: Specialization of matrix classes inherited from Matrix [3]

1.3 Modified dynamic programming algorithm for matrix-chain product

The following dynamic programming algorithm demonstrates how we handle different matrix types for the optimal associative ordering. A vector of pointers of matrix type is passed to this routine. The matrices could be any of a set of predefined types.

```
Matrix-chain-order(vp)
    n ← length[vp] - 1
    p ← getdimensions(vp)
    op ← 0
    tempOp ← 0
    for i ← 1 to n do
         m(f1(i,i)) ← 0
         tempS(f1(i,i))← 0
    for l ← 2 to n do
       for i ← 1 to n - l + 1 do
         m(f1(i, j)) ← INT_MAX
         for k ← i to j - 1 do
           q ← f2(m(f1(i,k)))+f2(m(f1(k+1,j)))+
               f3(p[i-1],[j],[k])
           if(q < m(f1(i, j))) then
               m(f1(i, j)) ← q
               s(f1(i, j)) ← k
               op ← q
            end if
            tempS(f1(i, j)) ← k
            tempOp ← q
    return m, s,tempS,op,tempOp
```

The complete internal details are not provided here in the psuedocode. The values of `m` and `s` are stored in a *packed* storage format. Packed storage is a space-efficient way to store elements in a single dimension array by eliminating zero elements [5]. Function f1 is used to calculate the index to store the non-zero elements into a packed array. Function f2 is used to identify the type of matrix. Function f3 calculates the number of scalar operations using any of the Equations (1), (3) or (5).

1.4 Results

Testing the algorithm with different combinations of general, dense-diagonal and dense-triangular matrices, we produced the following results which demonstrate that the optimal associative matrix-chain product ordering can change for fixed matrix dimensions when matrix specialization is exploited. The result using specialization is compared to the result using no specialization. Consider the sequence of four matrices

$$A_G^{1x3} B_D^{3x4} C_G^{4x3} D_D^{3x3}.$$

The subscripts G and D denote general dense and diagonal matrix types. The optimal ordering for this sequence is $(A(B(CD)))$ with 10 scalar operations. Using the same sequence with the original version of algorithm resulted in the same optimal sequence $(A(B(CD)))$ but with 27 scalar operations.

Matrix Type	Sequence	Optimal Ordering	Operation Count
General	$A_G^{2x4}B_G^{4x4}C_G^{4x4}D_G^{4x2}$	$\left(A\left(B(CD)\right)\right)$	80
Specialized	$A_G^{2x4}B_D^{4x4}C_D^{4x4}D_G^{4x2}$	$(A\left((BC)D\right))$	28

Table 1: General vs. Specialized matrix types

Table 1 shows different optimal orderings for a fixed set of matrix dimensions of general and specialized version. We observe that the matrix-chain product can be performed more efficiently when we can specialize the matrix types.

2 Conclusions

We provided a class method in C++ which implements the standard algorithm for optimizing the matrix chain product for general matrices. We then augmented that facility to exploit different fundamental dense matrix shapes. We provided examples where the optimal ordering changes when considering specific shaped rather than general matrices. The implementation is carried in the following steps. The first step provides the essential matrix-chain operand characteristics to a dynamic programming based solver which accepts a variable number of arguments and computes an optimal associative ordering. Then the product can be performed in the order as specified by the algorithm.

2.1 Further Developments

At our present stage of development, there are unresolved issues relating to whether some types of shaped matrices (e.g. diagonal and triangular) can be non-square. Our implementation can be extended to the remaining matrix types in the BLAS then to the additional matrix types as listed in [1]. Additionally, matrix-chain multiplication can be performed at compile time by the application of the template facility in C++. A programmer could then code the matrix-chain as an expression. The matrix-chain expression could then be preprocessed at compile time to generate the optimal matrix-chain ordering for the run-time computation of the product thereby increasing efficiency [3]. This is termed as *metaprogramming*. The matrix-chain expression can be evaluated by a technique called e*xpression templates* [11] which is implemented by overloading binary operators [6].

References

[1] Anda, A.A, *Generative Programming Considerations for the Matrix-Chain Product Problem*, Proceedings of the Midwest Instruction and Computing Symposium (MICS-08), March 2008, pp. 272-279.

[2] CORMEN, T.H., LEISERSON, C.E., R.L., AND STEIN, C. *Introduction to Algorithms*, Second edition, The MIT Press, Cambridge, MA, USA, 2001.

[3] CZARNECKI, KRZYSTOF, *Generative Programming: Methods, tools and Applications*, Addison – Wesley, Boston, MA, USA 2000.

[4] Template Numerical Tool Kit, http://math.nist.gov/tnt/

[5] Packed Storage Format, http://www.netlib.org/lapack/lug/node123.html

[6] Bassetti, Federico and Davis, Kei and Quinlan, Dan, *C++ Expression Templates Performance Issues in Scientific Computing*, Rice University, Texas, October 1997.

[7] Baase, Sara and Van Gelder, Allen, *Computer Algorithms: Introduction to Design and Analysis*, Third edition, Addison-Wesley, Reading, MA, 2000.

[8] Purdom, Jr., Paul Walton and Brown, Cynthia A., *The Analysis of Algorithms*, Holt, Rinehart and Winston, New York, NY, 1985.

[9] Neapolitan, Richard and Naimipour, Kumarss, *Foundations of Algorithms*, D. C. Heath and Company, Lexington, MA, 1996.

[10] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 16 ,1990.

[11] Veldhuizen, Todd L., *C++ templates as partial evaluation*, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 1999.