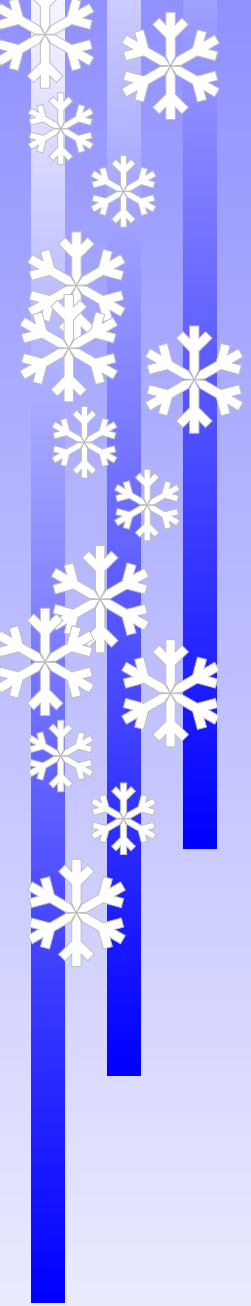

Generative Programming Considerations for the Matrix-Chain Product Problem

Andrew A. Anda, Ph.D.

aanda@stcloudstate.edu

Computer Science Department

St. Cloud State University

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ Summary

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ Summary

Overview

When specialists in a discipline / application domain implement a computational solution to a problem in that domain, they

1. specify, analyze, and formulate an algorithm
2. encode that algorithm $\Rightarrow$ a programming environment
   - ✓ application domain syntax/semantics $\Rightarrow$ programming language environment

Greater correlation: application $\Leftrightarrow$ programming domains,
$\Rightarrow$ Greater efficiency & reliability of translation process.

Overview [2]

- ✓ high-level programming language evolution ⇒ programmer *deskilling* via routine task automation ⇒ higher levels of interface abstraction;
- ✓ reasonable, common, & expected (overridable) defaults ⇒ programmer insulation from many implementation details;
- ✓ early high-level programming languages targeted specific broad application domains: e.g. COBOL & FORTRAN;
- ✓ *then* more specific domain expression via source libraries of related modular subprograms: e.g. BLAS & LINPACK;
- ✓ *then* gen.-purpose lang. ⇒ ext. facilities for domain specialization;
- ✓ OOP class/ADT: specialized domain knowledge & operability.
- ✓ higher abstraction level (often) ⇒ slower program performance

Overview [3]

- ✓ SIMULA67 ⇒ C ⇒ C++;
- ✓ classes ⇒ operator overloading ⇒ templates;
- ✓ operator overloading ⇒ furthuring the ongoing goal of matching program syntax/semantics to application domains;
- ✓ *however* op. overloading in C++ restricted;
- ✓ templates ⇒ generic programming,
  but *accidentally* ⇒ static (*potentially recursive*) computation ⇒ Turing-completeness, i.e. *meta-programming*.

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ Summary

The Matrix Arithmetic Domain

- ✓ Frequently, computational problems formulated via matrix algebra;
- ✓ Program matrix facilities better match their domain if the arithmetic operators are overloaded.
- ✓ (APL)
- ✓ $\Rightarrow$ intrinsic in Fortran 90 and its successors;
- ✓ $\Rightarrow$ extrinsic classes in C++ $\Rightarrow$ extensibility;

The Matrix Arithmetic Domain [2]

METAPROGRAMMING:

- ✓ C++ templates ⇒ static specializations that allow arith. ops. to abstract commonality, w. min. code dup., uniformly or specially;
- ✓ ⇒ *generative* static special case & condition handling;
- ✓ ⇒ expression evaluation;
- ✓ ⇒ execution code optimization (e.g. temporary reduction or loop unrolling);
- ✓ Early matrix class libraries used templates only for generics.
- ✓ Contemporary libraries (e.g. Blitz++, POOMA, MTL, and GMCL) optimize run-time performance on categories of matrix types.

The Matrix Arithmetic Domain [3]

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

The Matrix Arithmetic Domain [4]

- √ Sans generics, handling all matrix types ⇒ exp. combinatorial bloat;
- √ templates efficiently manage this complexity automatically:
- √ GMCL handles 1840 matrix types via ∼7500 LoC.

Matrix types & features are handled by a *configuration generator* called by an *expression generator* wherein overloaded ops. construct expression objects rep. the structure of expressions they're used in.

The assignment op. '=' handles the evaluation of the expression object tree.

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ MOTIVATION
- ✓ Proposal
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ Summary

Motivation

Typical linear/matrix algebra math courses present properties of matrix arithmetic exist, e.g. commutivity, distributivity, & associativity.

In general, for multiplication, matrices don't commute.

But they do associate:

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.

E.g. $(A * B) * C = A * (B * C)$.

However, seldom is presented the property that associativity *fails* w.r.t. operation counts for some matrix-product chains having non-square elements.

Motivation [2]

- ✓ Specialists, in domains unrelated to comp. sci.& combinatorics, would likely not know that by relying on the default L-R associativity of the * operator, they might be performing an excessive and sub-optimal number of scalar products.
- ✓ $\Rightarrow$ this instance of specialized matrix domain knowledge should be integrated into a generative matrix library.
- ✓ This goal is the focus of this proposal.

Motivation [3]

- ✓ $\exists$ optimal (*fewest operations*) orderings;
- ✓ optimal ordering determination is infeasible for longer chains, as the count of possible orderings grows exponentially with chain length proportional to Catalan numbers $\Omega(4^n/n^{3/2})$;
- ✓ dynamic programming via memoization of optimal subproblems permits solution in cubic time and quadratic space complexity.
- ✓ Hu and Shing reduced that to $O(n \log n)$ time, $O(n)$ space.
- ✓ even lower complexity algorithms exist but require parallelization or approximation.

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ PROPOSAL
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ Summary

Proposal

The preceding discussion 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 product using that ordering,

via C++ template metaprogramming.

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ CONSIDERATIONS
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ Summary

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ Considerations
    - → STATIC/DYNAMIC
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ Summary

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.
- ✓ the full optimization can occur statically only if **all** necessary parameters are determined at compile-time.
- ✓ fully static subchains can be processed at compile-time.

Static/Dynamic [2]

- The matrix-chain class will use a default optimal order alg.;
- Most chains will be short, so the standard alg. will suffice;
- More efficient ordering algs., if part of the class, could either be selected via user declaration, or auto. based on chain length;
- A user could select an approximating algorithm;
- A facility whereby the user may optionally provide their own algorithm via functor would add desirable extensibility.

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ Summary

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 pair of fully dense and general rectangular matrices.
- ✓ However, more specific characterizations of any of the two operands of a binary matrix product, such as those itemized in the far-from-exhaustive list of matrix types, can often be specified;
- ✓ $\Rightarrow$ those characteristics can be exploited by more efficient algs. (op. count/exec. time)
- ✓ Even gen. matrices can be mult. more effic., and with lower computational complexity,(if square) via one or more recursive applications of the Strassen-Winnograd matrix product algorithm.
- ✓ Similarly, the more effic. 3M algorithm can be used for gen. complex matrix products.

Binary Product Operation Counting [2]

- ✓ If accurate op. counts for the products of special matrices and algorithms are provided to the ordering optimizer, then optimal orderings may change.
- ✓ $\Rightarrow$ 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.
- ✓ operation counts could be generated *a priori*, or alternatively, benchmarking can be used to generate functions which estimate runtime performance. (*vis-a-vis* ATLAS)
- ✓ NUMA $\Rightarrow$ op. count $\neq$ performance.

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → IMPLEMENTATION ENVIRONMENT
    - → Beyond matrix-chain products
- ✓ Summary

Implementation Environment

- Rather than start from scratch, this project would be more feasibly performed by augmenting an existing metaprogrammed C++ matrix library. (e.g. GMCL)
- Although, a stand-alone class could be feasible and practical if it's restricted to ordinary dense general matrices and the standard product algorithm.
- This might be a good target for a *proof-of-concept* project.
- Certain essential generator and type related facilities have to be performed rather awkwardly and opaquely in the current C++ standard.
- **C++0X** will add critical syntax and semantics (e.g. *concepts*) that will permit a more elegant implementation of generic and generative metaprogramming techniques.

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → BEYOND MATRIX-CHAIN PRODUCTS
- ✓ Summary

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

1. $\sum_{i=1}^{n} a_i X^i$
2. $\sum_{i=1}^{n} A_i x^i$
3. $\sum_{i=1}^{n} A_i X^i$

can be evaluated efficiently via Horner's method in conjunction with the binary power algorithm. However, equation 1 can be evaluated more efficiently than by Horner's method.

Beyond Matrix-Chain Products [2]

- ✓ In fact, any function of a matrix can be extracted and evaluated with some being identified as having more efficient solution algorithms known.
- ✓ 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.

Outline

- ✓ Introduction
  - ⋆ Overview
  - ⋆ The Matrix Arithmetic Domain
- ✓ Motivation
- ✓ Proposal
  - ⋆ Considerations
    - → Static/Dynamic
    - → Binary product operation counting
    - → Implementation environment
    - → Beyond matrix-chain products
- ✓ SUMMARY

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.