A Systematic Approach to the Design and Analysis of Linear Algebra Algorithms

by

John Andrew Gunnels, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2001

Abstract

Over the last two decades, much progress has been made in the area of the high-performance sequential and parallel implementation of dense linear algebra operations. At what time can we confidently state that we truly understand this problem area and what form might evidence in support of this assertion take? It is our thesis that if we focus this question on the *software architecture* of libraries for dense linear algebra operations, we can claim to have reached the point where, for a restricted class of problems, we understand this area. In this dissertation, we provide evidence in support of this assertion by outlining a systematic and partially automated approach to the derivation and high-performance implementation of a large class of dense linear algebra operations.

We have arrived at a conclusion that the answer is to apply formal derivation techniques from Computing Science to the development of high-performance linear algebra libraries. The resulting approach has resulted in an aesthetically pleasing, coherent code that facilitates performance analysis, intelligent modularity, and the enforcement of program correctness via assertions. In this dissertation, we illustrate this observation by looking at the development of the Formal Linear Algebra Methods Environment (FLAME) for implementing linear algebra algorithms.

We believe that traditional methods of implementation do not reflect the natural manner in which an algorithm is either classified or derived. To remedy this discrepancy, we propose the use of a small set of abstractions that can be used to design and implement linear algebra algorithms in a simple and straightforward manner. These abstractions may be expressed in a script language that can be compiled into efficient executable code. We extend this approach to parallel implementations without adding substantial complexity.

It should also be possible to translate these scripts into analytical equations that reflect their performance profiles. These profiles may allow software designers to systematically optimize their algorithms for a given machine or to meet a particular resource goal. Given the more systematic approach to deriving and implementing algorithms that is facilitated by better abstraction and classification techniques, this sort of analysis can be shown to be systematically derivable and automated.

Contents

Chapter 1

Introduction

Our claim is that it is possible to create a system wherein one can code dense linear algebra routines in a very high-level, domain-specific language and still attain near-peak performance on distributed-memory parallel architectures. This dissertation provides evidence supporting this claim and describes the implications of such a system. Our thesis can be expressed as follows:

- We have discovered how to systematically derive a restricted class of linear algebra algorithms using formal derivation techniques.
- For this class of algorithms, compiler tools can be employed to reduce a domain-specific program to a list of operational requirements.
- In this domain, requirements can be paired to the functionality provided by a set of library routines if the annotations used to express those services are compatible with the requirements.
- For this class of algorithms, performance estimates of constructed routines can be made highly accurate if the underlying library is layered correctly and the language used to describe performance characteristics is suitably flexible.

The domain under study in this dissertation is restricted to a subset of dense linear algebra problems. This class includes the level-3 BLAS routines [25, 39], matrix factorization routines [44], and kernels involved in control theory [65, 64]. While this set of algorithms does not cover the gamut of dense linear algebra, it does comprise a useful, core set.

This chapter begins with an historical overview that summarizes the evolution of linear algebra software libraries. This is followed by a brief treatment of the insights that led us to the work presented here. We then explain how this work advances the state-of-the-art. After itemizing the contributions of our research, we present a summary of other research efforts whose goals are similar to our own. The final section of this chapter presents an outline of the dissertation.

1.1 Motivation

Advances in software engineering for scientific applications have often been led by techniques developed for libraries for dense linear algebra operations. The first such package to achieve widespread use and to embody new techniques in software engineering was EISPACK [68]. The mid-1970s witnessed the introduction of the Basic Linear Algebra Subprograms (BLAS) [55]. This version of the BLAS was a set of vector operations (now known as level-1 BLAS) that allowed libraries to attain high performance on computers possessing a flat memory while remaining portable between platforms. This library and its well-defined interface simultaneously enhanced code modularity and readability. The first successful library to exploit these BLAS was LINPACK [22].

By the late 1980s, it was recognized that in order to overcome the gap between processor and memory performance on modern microprocessors it was necessary to reformulate matrix operations in terms of level-2 (matrix-vector multiplication) and level-3 (matrix-matrix multiplication-like) BLAS operations [26, 25]. First released in the early 1990s, LAPACK [5] is a high-performance package for linear algebra operations. LAPACK is a portable library that provides a functionality that is a superset of both LINPACK and EISPACK. The LAPACK library heavily utilizes the level-3 BLAS and evinces high performance on essentially all sequential and shared-memory architectures.

A major simplification in the implementation of the level-3 BLAS stemmed from the observation that they can be cast in terms of optimized matrix-matrix multiplication [1, 47, 52]. The performance of the resulting libraries was comparable to that of the optimized, assembly-coded, vendor-supplied BLAS in many cases. Further, the implementations were more portable than previous BLAS libraries because they were written in Fortran. In those cases where the code was not performance transportable (i.e. where these BLAS did not compile into efficient assembly code), the ideas behind this research simplified the task of hand-coding the level-3 BLAS library.

With the advent of distributed-memory parallel architectures, LAPACK was no longer sufficient for the needs of high-performance scientific computing. LAPACK worked well with high-performance shared-memory systems, but was not written to be compatible with distributed-memory architectures. Distributed-memory architectures depend upon the applications and libraries to explicitly manage the physically distinct memories attached to the computational processors (nodes) of the system. Thus, a parallel version of LAPACK, ScaLAPACK [15], was developed. A major design goal of the ScaLAPACK project was to preserve and re-use as much code from LAPACK as possible. Thus, all layers in the ScaLAPACK software architecture were designed to resemble analogous layers in the LAPACK software architecture. This decision was motivated by the fact that LAPACK had proven itself both robust and efficient. However, this decision complicated the implementation of ScaLAPACK. The introduction of data distribution across memories created a complication analogous to that of creating and maintaining the data structures required for storing sparse matrices. The mapping from indices to matrix element(s) was no longer a simple one. Combining this complication with the monolithic structure of the software led to code that was laborious to construct and difficult to maintain.

Recently, a number of projects have developed software for generating automatically tuned matrix-matrix multiplication kernels. These undertakings include the PHiPAC project [11] and the ATLAS project [76].

The PHiPAC research effort included a careful analysis of C implementations of matrix-matrix multiplication. By structuring the loops and memory references carefully, it is possible for a C compiler to generate highly efficient code for this algorithm. The PHiPAC research team produced a software system capable of generating efficient BLAS kernels through a generate-and-test strategy. This software generator created implementations of matrix multiplication algorithms that blocked matrices in every reasonable way. By executing these programs and monitoring the resulting performance, parameters for a high-performance matrix multiplication implementation could be determined.

The ATLAS project repackaged and simplified the methods developed in creating the PHiPAC system. In addition, the ATLAS system required less time to generate efficient linear algebra kernels. This efficiency was gained by avoiding PHiPAC's exhaustive search of the parameter space involved in determining optimal matrix blocking sizes. Unfortunately, as this search space was reduced through experience, not by a theoretical model, it is sometimes the case that ATLAS produces code with far less than optimal performance characteristics [42].

1.2 Our Approach

1.2.1 Recent Insights

The primary inspiration for much of the work presented in this dissertation came from our experience with the Parallel Linear Algebra Package (PLAPACK) [74]. PLAPACK achieves a functionality similar to that of ScaLAPACK, targeting the same distributed-memory architectures. In contrast to ScaLAPACK, PLAPACK uses an MPI-like [38] approach to hide indexing and data distribution details.

Work related to PLAPACK provided insights that motivated the approach presented in Chapter 2 and Chapter 3 of this document. Raising the level of abstraction at which one codes reduces the effort involved in implementing high-performance linear algebra library routines.

As we gained more experience with PLAPACK, a number of themes kept reappearing:

- The derivation of algorithms for different linear algebra operations was systematic.
- Similarly, the analysis of the resulting algorithms was systematic, although tedious and error-prone.
- For a given linear algebra operation, different algorithms provided better performance as the sizes of operands (matrices) changed [40]. This makes analysis necessary in order to be able to determine when and understand why different algorithms are superior.

We discovered that, in deriving algorithms for a new operation, we were applying formal derivation methods to the domain of algorithms for dense linear algebra operations. This led to our work on the Formal Linear Algebra Methods Environment (FLAME), research detailed in Chapter 2.

Linear algebra libraries are expected to contain routines that can deal with a broad range of operational tasks and to be written in a form that can be ported between different computational environments. The LAPACK library achieves both objectives by exploiting the BLAS. However, the use of libraries such as LAPACK has the disadvantages of requiring the applications programmer to perform time-consuming, involved, source code optimizations that are often not performance portable [50]. The work presented in Chapter 3 and Chapter 4 addresses this problem. By creating a language that allows the user to program at a level of abstraction higher than that of PLAPACK, little library knowledge is required of the programmer. An automated code generation system accepts programs written in this language and produces code that evinces superior performance on distributed-memory, parallel supercomputers. This is achieved by mechanically linking the high-level programs to a functionally-annotated version of the PLAPACK library.

A simple model of a distributed-memory parallel system is used for performance analysis in Chapter 5. This model reflects lessons learned while studying the issues related to the creation of high-performance matrix-matrix multiplication kernels for single processor machines with hierarchical memories [42]. This contrasts with code generation efforts such as PHiPAC and ATLAS, which employ brute force to search a parameter space for blocking sizes that accommodate multiple levels of memory hierarchy.

Together, these experiences and insights led us to conclude that for a subset of dense linear algebra operations, the derivation, implementation, and analysis of parallel algorithms is now a well-understood and systematic process.

1.2.2 A Solution: The Big Picture

The goal of linear algebra code production is to generate efficient code from a clear statement of mathematical requirements. Our strategy for achieving this objective is depicted in Figure 1.1. Specifically, it is our aim to replace the "Human Expert" of Figure 1.2, which reflects where previous research had led us, with systematic techniques and automated tools. The term "efficient" covers a number of sub-goals including reliability, speed, and transportability. These qualities are widely considered the primary value metrics of such computer codes. This dissertation targets the community of scientific library writers. Since one might safely suppose that these researchers are mathematicians or have strong mathematical backgrounds, the clear statement of mathematical requirements is a logical starting point. The mathematical specification of the problem must be known in order to generate code to solve that problem. In order to automate a system, this specification, represented by "A = LU" in Figure 1.1, must be made explicit.

The unified approach to the design and development of dense linear algebra algorithms that is presented in this document should be distinguished from the situation wherein development is *ad hoc*. When the development and tool sets are collected, not designed as