The Tailoring of Abstract Functional Specifications of Numerical Algorithms for Sparse Data Structures through Automated Program Derivation and Transformation

Stephen Fitzpatrick, M. Clint, T. J. Harmer and P. Kilpatrick

Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN,
Northern Ireland, U.K.
{S.Fitzpatrick, M.Clint, T.Harmer, P.Kilpatrick}@cs.qub.ac.uk

Abstract

The automated application of program transformations is used to derive, from abstract functional specifications of numerical mathematical algorithms, highly efficient imperative implementations tailored for execution on sequential, vector and array processors. Emphasis is placed on transformations which tailor implementations to use special programming techniques optimized for sparse matrices. We demonstrate that derived implementations attain superior execution performance than manual implementations for two significant algorithms.

1 Introduction

Scientific computing is dominated by matrix and vector algorithms, drawn from linear algebra. Many such algorithms can be formally defined in a simple, elegant manner that is independent of any particular computational architecture and that is amenable to analysis. Yet their efficient implementation — and efficiency is normally paramount in scientific computing — usually requires a low-level, architecture-specific formulation in a programming language (Fortran or C) that possesses ill-defined semantics.

It is thus generally the case that an efficient implementation of a numerical algorithm is almost completely different in form from any high-level definition of the algorithm — the form of an implementation is dictated as much by the computational architecture as by the computations required by the algorithm. The lack of a ready correspondence between definition and implementation can make an implementation difficult to construct initially, difficult to verify once constructed and difficult to maintain as the algorithm and implementation techniques evolve. The strong influence of the computational architecture on an implementation often hinders adaptation for another architecture.

Furthermore, linear algebra algorithms can usually be applied to arbitrarily large data sets; practical applications often require data sets for which standard programming techniques would result in an implementation that requires too much storage or that takes too long to execute to be useful. In such cases, it may be possible to use special programming techniques based on properties of the data sets (such as symmetry or sparsity) to reduce the time and storage requirements of an implementation.

While such programming techniques are often conceptually simple, their use normally requires considerable attention to detail from the programmer (so their use is error prone) and often distorts an implementation of an algorithm even further from any high-level definition of the algorithm (so further hindering construction, verification and maintenance).

This paper demonstrates how a programmer can construct clear, high-level, architecture-independent algorithm specifications using a functional programming language, and yet still obtain optimal execution performance by deriving architecture-specific, low-level, imperative implementations through the automated application of program transformations. In particular, *this paper discusses how sparse programming techniques can be automatically incorporated into an implementation*. The quality of derived implementations of two non-trivial algorithms is illustrated by demonstrating execution performance superior to that attained by manual implementations.

The formal properties enjoyed by pure functional languages are crucial in the derivation of high-performance implementations and to the adaptation of implementations for sparse data. It is thus hoped that this paper will appeal to a wide spectrum of readers: those interested in formal properties of languages, those interested in programming techniques for large data sets and those interested in producing highly efficient implementations.

1.1 Program Derivation

The method of formal program derivation can be used to overcome the problems arising from the discrepancy in form

between algorithm definition and algorithm implementation. Rather than manually construct an implementation that is intended to correspond to the definition, an implementation is derived step-by-step from the definition, each step of the derivation being performed by the application of a correctness-preserving program transformation. A derived implementation is thus known by construction to match the definition.

In the work reported here, program transformation and derivation is entirely automated: a programmer constructs transformations using a formal, wide-spectrum grammar; arranges the transformations in the order they are to be applied; and initiates the derivation process, giving a definition of the algorithm that is to be implemented. The programmer's rôle is thus to develop methods that can be used to construct implementations, and to encode these methods as sequences of transformations.

The automated approach to program derivation overcomes the problems of construction, verification and maintenance:

- If the transformations and the transformation system are trusted, the derived implementation can be assumed to match the source definition.
- If an algorithm definition is changed, simply re-derive an implementation. The process is automated, so little effort is required from the programmer.
- Evolution of an implementation method requires changing and/or extending a subset of the transformations. New implementations can then be derived automatically.
- An automated system is not prone to human failings such as lapses in concentration when using intricate implementation methods. A programmer develops systematic methods for implementing algorithms and encodes these methods as transformations; the transformation system attends to the mechanical details of applying the rules.

1.2 Families of Implementations

Experience has shown that many transformations are independent of the computational architecture for which an implementation is being derived. Derivations may require relatively minor modification to produce implementations for other architectures. Program derivation thus permits a family of architecture-specific implementations to be produced from a single, architecture-independent specification.

In previous work, Boyle, Harmer, Fitzpatrick *et. al.* [1, 2, 3] have discussed the derivation of implementations for sequential, vector, array and multi-processor systems. In this paper, an additional stage for these derivations is discussed. This new stage can optionally be applied to tailor implementations for a particular form of data — sparse matrices. That is, a programmer produces an algorithm definition which uses standard (dense) definitions of operations. He modifies this definition only by adding assertions to specify the type of sparsity of 'input' matrices. The new stage of the derivations optimizes computations to take advantage of sparsity and introduces a compact storage scheme for the sparse data.

The sparse optimizations are independent of the target architecture, so the sparse stage can be combined with stages that produce sequential, vector or array implementations. These architecture specific stages are the same stages that would be applied even were the sparse stage not applied, in which case they produce dense implementations. The use of sparse programming techniques is thus considered here as just another implementation method, orthogonal to other implementation methods. Merely by choosing an appropriate combination of stages, a programmer can produce dense or sparse implementations with little additional effort. This is in marked contrast to the traditional implementation methods, which appear to treat the development of a sparse implementation as a problem entirely separate from that of developing a dense implementation.

In this paper, we do not consider algorithms which *construct* sparse matrices (for example, an algorithm which computes the product of two sparse matrices). We consider only algorithms in which the sparsity occurs in 'input data'. The method reported here can readily be extended for at least some simple cases of sparse construction, but extension to more complex construction may be more difficult. For example, normalizing a sparse matrix (so that the sum of the squares of its elements is 1) is a simple form of sparse construction that can be detected and optimized automatically; reordering the rows and columns of a sparse matrix to reduce the band width is a form of construction that may be difficult to treat automatically.

Indeed, it may be that some methods of construction, such as reordering, are used only for sparse versions of algorithms (with no equivalent construction being performed in dense versions). In such cases, it may be more accurate to consider the sparse and dense versions as being implementations of different algorithms, rather than as specializations of the same algorithm. However, it is worthwhile investigating the proposed methods to determine the circumstances in which they are useful.

1.3 Overview of Paper

In this paper, derivations are discussed for several types of sparse matrices: tridiagonal, symmetric tridiagonal, general banded and row-wise (a fixed number of non-zero elements in each row). Implementations are derived for sequential, vector and array architectures. The derivations are illustrated using matrix product; two more complex examples are also considered.

For the examples, it is assumed the reader possesses some familiarity with vector and array processors (particularly the CRAY and AMT DAP), though no knowledge of these is required for the main discussion.

The paper is structured as follows:

Section 2

Functional specification of algorithms and the basic functions used to define array operations.

Section 3
Program derivation and transformation; how derivations are structured to permit families of implementations to be produced.

Section 4
Details of the derivation for tridiagonal matrices.

Section 5 Extension for symmetric tridiagonal matrices.

Section 6 Extension for higher-banded matrices

Section 7 Extension for row-wise sparse matrices.

Section 8 Results: assessment of derived implementations.

Section 9 Related work.

Section 10 Conclusions.

Some readers of this paper will find a number of sections all too familiar: functional programming experts may wish to skip section 2; those familiar with program transformation or formal rewrite rules may wish to skip section 3.

2 Functional Specifications

The derivation of an implementation of an algorithm requires a formal definition of the algorithm suited to automated manipulation. Most mathematical definitions of algorithms consist of a mixture of formal notation and stylized English text, and so are not quite suited as sources for derivations.

In this paper, *functional specifications* of algorithms are used. Functional specifications are essentially recastings of mathematical definitions into the entirely formal notation of a functional programming language — here, the SML (Standard Meta Language [4]) programming language is used. Functional specifications retain the high-level, machine-independent nature of mathematical definitions. In addition, they are usually easy to learn by anyone with a basic mathematical background.

One of the primary characteristics of functional programming languages is that they are expression based (just as mathematics is expression based): a functional program is an expression; execution of the program means evaluating the expression. This expression basis provides a much simpler semantics than the state-based semantics of imperative languages, facilitating human understanding and analysis of programs, and the automated manipulation of programs (e.g. program transformation).

For example, the matrix transpose operation could be defined as follows: The transpose A^T of an $m \times n$ matrix A is an $n \times m$ matrix with elements

$$A^T_{ij} \equiv A_{ji} \ .$$

A functional specification of transpose is:

```
fun transpose(A:α matrix,m:int,n:int):α matrix
  = generate([n,m],fn(i:int,j:int)=>A@[j,i])
```

This specification defines a function for performing matrix transposition. The function constructs and returns the transpose of its matrix argument. Since there is no concept of 'state' in a functional language, there is no question of a function overwriting its arguments or causing other 'side-effects': all functions are *pure*. Pure functions provide a superior decomposition technique to subroutines or 'functions' in imperative languages, since pure functions have a much narrower interface: the only effect of a function application is to compute a value, and all the parameters that affect the computed value are passed as explicit arguments to the function.

The transpose of the matrix is constructed using the generate function.

- The notation [a, b, ...] denotes a homomorphic list having elements a, b, etc. Lists are used in this paper to specify, according to context, either a single array index or an array domain (a set of indices); for example:
  - as an array index, the list [3, 5] indicates that element, of a two-dimensional array, occuring at the third position in the first dimension and at the fifth position in the second dimension;
  - as an array domain, the list [m, n] denotes the index set $\{[i,j] : 1 \leq i \leq m, 1 \leq j \leq n\}$.

  List notation should not be confused with the traditional notation used when displaying a matrix, such as in Figure 4.
- The first argument to generate gives the dimensions of the matrix being constructed.
- The second argument is a function, called the *generating function*, which takes two integers (i and j) as arguments. The generating function defines the elements of the matrix: the value of element (i', j') is the value of the generating function applied to i' and j'.
- The body of the generating function is A@[j,i] (read "A at (location) j,i"). The @ symbol is an operator which performs array indexing; thus, A@[j,i] is (the value of) element (j, i) of A.

(The array functions and operators used in this paper are not part of SML — they are part of a library of operations developed by the authors to assist the specification and transformation of algorithms for scientific applications.)

An important aspect of the generate function is that no order is defined for constructing the elements: no order need be defined as, in the context of a pure functional language, the construction of one element cannot affect the construction of any other element. The elements can thus, for example, be constructed simultaneously if that is appropriate for some particular computational architecture.

Definitions such as the one above are referred to as *specifications* to emphasize that the definitions convey *what is*

to be computed, rather than *how the computation is to be performed*; that is, primary importance is attached to the *denotational meaning* of a specification rather than to its *operational meaning*. For example, the 'program' x+y has denotational meaning 'the sum of (the value of) x and (the value of) y' and operational meaning 'evaluate x; evaluate y; perform the addition operation'.

In concrete terms, our emphasis on the denotational meaning requires that as an implementation is derived from a specification, its denotational meaning must be preserved, but its operational meaning may change. Derivations use properties of the denotation to modify a specification into forms that have operational meanings better suited to some purpose (for example, to execution on an array processor, or to computations on sparse matrices). *The denotational meaning provides the formal basis for modification while the operational meaning provides motivation to guide modification*.

2.1 Specification of Matrix Product

As a more complex example of a functional specification, consider the computation of the product of two matrices. The simple 'row into column' definition may be formalized as shown in Figure 1.

- The function mmmult computes the product of two matrices, A and B. The product is constructed using generate. The dimensions of the product are defined using size, which returns the size of an array in the specified dimension (thus, size(A,1) is the number of rows in A).
- Each element of the product is defined as the inner-product of a row of A and a column of B (the definitions of the row and column functions are straightforward applications of generate).[1]
- The inner-product of two vectors is defined using reduce, which combines a set of values into a single value by the repeated application of a binary *reducing function* (here, real addition).The set of values to be reduced is defined by the first and second arguments to reduce: the first is a list of dimensions specifying a set of indices; the second is a generating function which is applied to each index in that set to provide the values that are to be reduced. Thus, the inner-product may informally be regarded as the sum:

$$\Sigma_{i=1}^{\text{sizeA}} A@[i] * B@[i] \quad .$$

The function definitions are straightforward recastings of standard mathematical definitions using a formal notation.

The common mathematical operations on vectors and matrices are provided for use in specifications as a library of

```
fun innerproduct(A:real vector,B:real vector):real
  = reduce([size(A)],fn(i:int)=>A@[i]*B@[i],+,0)

fun row(A:real matrix,i:int):real vector
  = generate(size(A,2),fn(j:int)=>A@[i,j])

fun column(A:real matrix,j:int):real vector
  = generate(size(A,1),fn(i:int)=>A@[i,j])

fun mmmult(A:real matrix,B:real matrix):real matrix
  = generate([size(A,1),size(B,2)],
    fn(i:int,j:int)=>innerproduct(row(A,i),column(B,j)))
```

Figure 1: SML specification of matrix product

functions. Most of the operations can be invoked using standard operator notation. For example, '+' denotes vector and matrix addition and left or right addition of a scalar to a vector or matrix (as well as integer addition and real addition). The '*' operator, when applied to two matrices, denotes algebraic matrix product rather than elementwise product.

3 Derivations and Transformations

Functional specifications of algorithms may be regarded as useful objects in themselves, as they facilitate understanding and analysis of algorithms. However, the primary theme of this paper is their use as sources from which practical, highly efficient implementations may be derived by program transformation. In this section, program transformations and the grouping of transformations into derivations, as realized in the TAMPR transformation system [5], are discussed.

3.1 Rewrite Rules and Transformations

A *program transformation* is a sequence of *rewrite rules*. Application of a rewrite rule changes each program section that matches a specified *pattern* into a specified *replacement*. For example, consider the rewrite rule

```
.sd.
 <entity>"1"+<entity>"1"
 ==>
 (2*<entity>"1")
.sc.
```

- The symbols .sd. and .sc. indicate the start and end of a rewrite rule.[2]
- The pattern is <entity>"1"+<entity>"1".
- The symbol <entity> is a non-terminal symbol in the wide-spectrum grammar that is used in this work; <entity> roughly corresponds to an expression.

[1] In functional specifications, a vector is a one-dimensional array. The usual mathematical distinction between 'row vectors' and 'column vectors' is disregarded, as this is based upon an identification between a vector and a matrix in which one dimension is of size 1.

[2] The symbols stand for 'structural definition' and 'structural change', respectively.

```
{
.sd.
  <entity>"1"*(<entity>"2"+<entity>"3")
  ==>
  <entity>"1"*<entity>"2"+<entity>"1"*<entity>"3"
.sc.
.sd.
  (<entity>"1"+<entity>"2")*<entity>"3"
  ==>
  <entity>"1"*<entity>"3"+<entity>"2"*<entity>"3"
.sc.
}
```

(a) The distributive transformations

```
(a+b)*(c+d)

==> by the second rule
a*(c+d)+b*(c+d)

==> by the first rule
a*c+a*d+b*(c+d)

==> by the first rule
a*c+a*d+b*c+b*d
```

(b) Example application of the transformations

Figure 2: Transformations for distributive operations

- The label "1" attached to the two <entity>s in the pattern indicates that the same expression must be matched by both <entity>s for the pattern as a whole to match. For example, the pattern would match f(x)+f(x), but not 1+2.
- The replacement is (2*<entity>"1").
- The occurrence of <entity>"1" in the replacement stands for whatever was matched by <entity>"1" in the pattern. Thus, f(x)+f(x) would be changed into (2*f(x)).

It is important to note that functional specifications are parsed according to a formal (wide-spectrum) grammar and that transformations operate on grammatical structures rather than on simple textual representations. For example, the expression 2*f(x)+f(x) is parsed as (2*f(x))+f(x); thus it cannot be matched by the pattern <entity>"1"+<entity>"1".

Figure 2(a) shows two rewrite rules for performing some basic distribution operations on expressions. When the TAMPR system applies a transformation to a program, it exhaustively applies all of the rewrite rules, until the program cannot be further changed by the transformation; i.e. when transformation is complete, no part of the transformed program can be matched by any of the patterns of the rewrite rules. For example, the expression (a+b)*(c+d) would be transformed as shown in Figure 2(b).

All of the rewrite rules used in the work reported here are defined using a formal grammar. However, in this paper, it is more convenient to use an informal notation, with names representing non-terminal symbols. For example,

```
.sd.
  <entity>"1"+<entity>"1"
  ==>
  (2*<entity>"1")
.sc.
```

may be written informally as

.sd. e+e ==> (2*e) where e is an expression .sc.

Qualification of the syntactic classes (such as 'where e is an expression') may be omitted when implied by context.

3.2 Derivations

A single transformation brings about some simple, well-defined change in a program. More complex changes are brought about by applying a *derivation*, or sequence of transformations; each transformation performs one of the multiple steps required to effect the overall change. The transformation system applies each transformation in the sequence once, in turn. (A single application of a transformation implies the *exhaustive* application of the rewrite rules that constitute the transformation.)

As an example of a derivation, consider β-reduction in the λ-calculus. In the λ-calculus, some expression, say e, can be bound to a name, say a, using the notation

$$\lambda a \cdot B\ (e)$$

The meaning of this expression is the expression B with all free occurrences of a replaced by (the value of) the expression e. (A free occurrence of a name is an occurrence of the name that does not appear within another binding which introduces the same name.) For example, the expression

$$\lambda a \cdot a+1\ (5)$$

has meaning 5+1.

The process of replacing a bound name with the value to which it is bound, is called β-reduction. If the restriction to free occurrences is ignored (i.e. if it is assumed that all names are unique) β-reduction can be performed by a simple transformation:

```
.sd.
  λx·e1 (e2)
  ==>
  e1 { .sd. x ==> e2 .sc. }
.sc.
```

in which the *sub-transformation* { .sc. x ==> e2 .sc. } is applied to only the expression e1 (rather than to the entire program).

It cannot generally be assumed that all names introduced in λ-bindings are unique. However, the name introduced by

a λ-binding is arbitrary, and can be systematically changed to any other name (provided the new name is not already in use). This is known as α-conversion. A transformation can be applied to perform α-conversion before β-reduction is performed.[3]

For practical purposes, it is often undesirable to β-reduce all bindings: if there is more than one occurrence of a bound name in an expression, β-reduction may result in the repeated evaluation of the expression to which the name is bound — this may degrade program performance if the expression is computationally expensive. It may be useful to identify certain types of expression that are not to be reduced: for example, expressions involving applications of recursive functions, or expressions that evaluate to non-scalar types.

Thus, a derivation to perform β-reduction may comprise the following steps:

Mark potentially expensive computations
: Notation is introduced to identify computations that may be expensive (based, say, on type information or analysis of function definitions). For example,

  f(x) ==> expensive(f(x))
  where f is a recursive function

  The pseudo-function 'expensive' designates an expensive computation; it is for use only in this derivation and is removed later.

Perform α-conversion
: This establishes the assumption used in the following transformation, that all names are unique.

Perform β-reduction
: β-reduce only those bindings in which the bound value does not contain an application of the expensive function.

Remove expensive
: 'Clean up' by removing the temporary notation used to designate expensive computations.

This example illustrates several points about the construction of derivations:

- A derivation is a sequence of clearly defined, relatively simple steps. It is preferable, where possible, to split a task into several transformations, rather than to implement the task as a single, complex transformation.

  In many ways, it is the identification of the steps that is the important aspect of constructing derivations: the transformations are the means of implementing the steps.

- It can be useful to introduce temporarily a notation to propagate information from points in a program where information is available to points where the information is to be used. In this example, the 'information' required is whether or not a function is recursive. This information must be obtained by analysis of the function definitions for the program. The information is used in expressions involving applications of the functions.

  Derivations often follow the pattern: localize information; use the information; remove the information.

[3] The β-reduction transformation could be used to perform β-reduction even in the presence of name clashes, if an appropriate application order is chosen. However, for this example, not all λ-bindings are reduced, so α-conversion is still required.

3.3 Sub-derivations and Intermediate Forms

For more complex derivations, it may be useful to introduce a further level of structure, by dividing a derivation into a sequence of *sub-derivations*, each of which is a sequence of transformations (and perhaps lower-level sub-derivations). For example, a derivation to convert SML into Fortran 77 may consist of two sub-derivations: the first to convert SML into an *intermediate form* such as the λ-calculus; the second to convert the intermediate form into Fortran 77.

SML ⟶ Fortran 77
≡ SML ⟶ λ-calculus
⟶ Fortran 77

In general, an initial form is converted into a final form through a sequence of intermediate forms, each intermediate form being produced by a sub-derivation. For example, the above derivation could be extended to optimize a program by performing function unfolding and common sub-expression elimination, in addition to performing straightforward conversion to Fortran.

SML ⟶ Fortran77
≡ SML ⟶ λ-calculus ⟶ Unfolded
⟶ Common Sub-expressions Eliminated
⟶ Fortran77

- Function unfolding is the replacement of each application of a (non-recursive) function with the function's definition. Function unfolding is performed for several reasons:

  – It reduces implementation overheads caused by the function calling mechanism.

  – Function definitions divide a program into conceptually simple segments. However, optimizations can often be performed only when the separate segments are brought together (for example, multiple loops can be combined into a single loop, reducing loop overheads). Function unfolding is perhaps the simplest method for bringing segments together.

  – A function definition is designed to be general and independent of the context in which a function is to be applied. (Since a function is usually applied in several different contexts, its definition must be context independent.) However, functions can

often be optimized based upon context (for example, based upon the arguments used in a particular application of the function). Function unfolding permits a function to be tailored precisely to each context in which it is applied (the tailoring being performed on the unfolded definition).

– Function unfolding eliminates all functions except a (small) set of primitive functions. These primitive functions are the only functions that need be considered when defining the semantics of data types and when considering program transformations (that are applied after unfolding): the semantics of all other functions follow from the semantics of the primitive functions combined with the semantics of function composition; since non-primitive functions no longer appear in a program, there is no need to consider what effect a transformation might have on them. For example, the only array functions considered in detail in this paper are the primitive functions element, generate and reduce.

Thus, function unfolding improves implementation efficiency and simplifies transformation.

- Common sub-expression elimination is the combining of multiple occurrences of a computationally expensive expression into a single occurrence. This combining can be expressed in the λ-calculus as a λ-binding.
- The final sub-derivation converts a functional form into an imperative form [2]. This sub-derivation is responsible for the translation of pure computations performed on values into equivalent, destructive manipulations of variables; for example, a conditional expression is translated into a conditional statement. This sub-derivation also performs some optimization on the re-use of variables, to eliminate at least partially the overheads associated with the maintenance of referential transparency in functional implementations.

By structuring a derivation into independent sub-derivations, a programmer can concentrate on each *implementation issue* (such as unfolding definitions) independently. Furthermore, converting the λ-calculus into Fortran is independent of the initial language: conversion is based upon the λ-calculus, not on SML. If other specification languages can be converted into the λ-calculus, then most of the derivation can be used unchanged with the other languages. For example, a derivation to convert Lisp into Fortran 77 is:

Lisp → Fortran 77
≡ Lisp → λ-calculus → Unfolded
→ Common Sub-expressions Eliminated
→ Fortran 77

Similarly, most of the derivation is independent of the final language (Fortran 77). The final sub-derivation can be replaced with another that converts the λ-calculus into, say, C.

SML → C
≡ SML → λ-calculus → Unfolded
→ Common Sub-expressions Eliminated
→ C

Alternatively, a sub-derivation can be introduced that converts array operations into an explicitly data-parallel form, followed by a sub-derivation that converts the data-parallel form into one of the array-based Fortran dialects such as Fortran Plus Enhanced (FPE) for the AMT DAP array processor.

SML → FPE
≡ SML → λ-calculus → Unfolded → Array
→ Common Sub-expressions Eliminated
→ FPE

Again, structuring a derivation permits implementation issues to be separated: converting to a data-parallel form is mostly independent of converting into the specific syntax of a particular programming language — the DAP sub-derivation could be replaced with one that converts the data-parallel form into, say, Fortran 90 or High Performance Fortran.

A further alternative for the final stage of the derivation is to convert the λ-calculus form into Fortran 77, but in a form suited to vectorizing compilers (in which loops can be vectorized).

SML → Fortran 77
≡ SML → λ-calculus → Unfolded → CRAY
→ Common Sub-expressions Eliminated
→ Fortran 77

The various intermediate forms produced by the sub-derivations can be represented as a family of forms, as shown in Figure 3.

Function unfolding and the derivation of imperative forms are discussed elsewhere [1, 2, 3, 6, 7]. This paper concentrates on a *sparse sub-derivation*; that is, a sub-derivation that tailors specifications for operation on sparse matrices. This sub-derivation is applied to the optimized functional form to convert standard definitions of functions into forms that take advantage of sparsity to reduce the amount of computation, and that use a compact storage scheme. The output of the sparse sub-derivation is still in functional form (i.e. it denotes computations as expressions). The imperative sub-derivations are applied to this form to produce implementations tailored for various architectures. (Note that the imperative sub-derivations do not need to be altered to accommodate the sparse sub-derivation.)

4 Implementation for Tridiagonal Matrices

The functional specification of numerical algorithms and the derivation of efficient implementations have previously been discussed. In this Section the derivation stage that optimizes specifications for tridiagonal matrices is discussed.

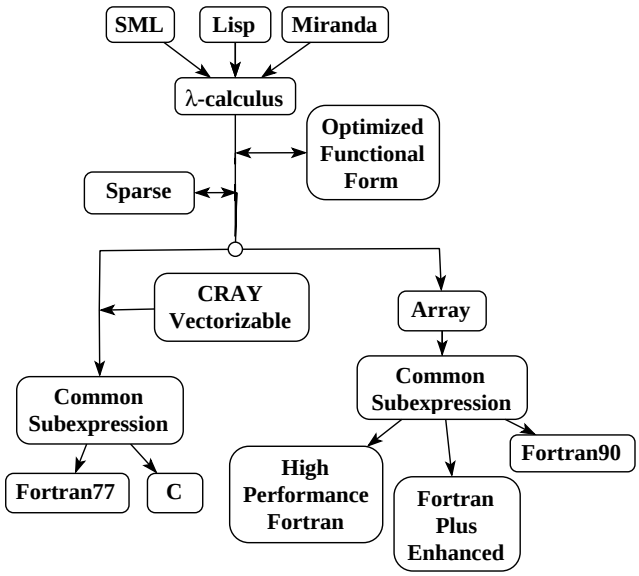

Figure 3: A family of intermediate forms

4.1 Sparse Matrices

A tridiagonal matrix is a particular form of *sparse matrix*: a matrix of real elements is said to be sparse if 'most' of its elements have value zero [8].[4] A sparse matrix is said to be *structured* if the non-zeros occur in a definable pattern (if their positions can be specified parametrically); otherwise a sparse matrix is *unstructured* (in which case the locations of the non-zeros must be individually specified).

There are many types of structured matrices (having their non-zeros in different patterns). Figure 4(a) shows one example, a tridiagonal matrix — that is, a matrix in which all of the non-zeros lie in the tridiagonal region (consisting of the main diagonal and the first sub- and super-diagonals; i.e. the region for which the row and column indices differ by not more than 1).

Figure 4(b) shows an example of an unstructured sparse matrix.

In this section, only tridiagonal matrices are considered; other forms of sparsity are considered later.

4.1.1 Reducing Storage Requirements

The storage required for a sparse matrix can be reduced by storing only the non-zero elements in a *primary store*; for an $n \times n$ tridiagonal matrix, the minimum storage requirement is reduced from n^2 units to $(3n-2)$ units.

There are several ways in which the non-zeros can be stored. The method used here is to store the non-zeros in a matrix of width 3: the first column stores the elements of the sub-diagonal, the second column stores the elements of the diagonal, and the third column stores the elements of the super-diagonal. See Figure 4(c).

This storage scheme is characterized by three mappings:

- the mapping from the tridiagonal region to the rectangular shape:

$$tridiagonal(n) \rightarrow [n, 3] \ ;$$

- the mapping from *sparse index* $[i, j]$ (which indicates a position in the $n \times n$ sparse matrix) to *primary index* $[i', j']$ (which indicates a position in the $n \times 3$ primary store):

$$[i, j] \rightarrow [i, j - i + 2]$$

and the inverse mapping:

$$[i', j'] \rightarrow [i', i' + j' - 2] \ .$$

The domain of the inverse is the set of indices represented by the rectangular shape $[n, 3]$ *excluding* the two indices $[1, 1]$ and $[n, 3]$. These two indices are *invalid* — they do not correspond to any location in the tridiagonal region. Since the primary store must have some value at these locations, it is assigned the value zero, allowing the invalid locations to be included in some, but not all, operations as if they were valid locations (for example, the invalid locations can be included in the summation of the elements without affecting the value of the sum).

4.1.2 Reducing Execution Time

The execution time of an implementation can be reduced by restricting operations to only the non-zero elements of a sparse matrix. For example, consider the computation of the product vector P of a matrix A and a vector V (all of order n): $P \equiv AV$. One element of P is the inner-product of a row of A with V:

$$P_i \equiv \sum_{k=1}^{n} A_{ik} V_k \ .$$

[4] The term 'most' is intentionally vague: a matrix may be considered sparse when it is worthwhile using special programming techniques to reduce the time and/or space requirements of an implementation.

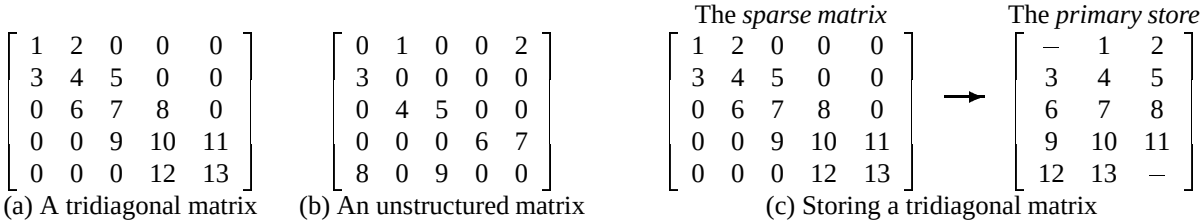

Figure 4: Sparse matrices

The computation of this inner-product requires n multiplications and $(n-1)$ additions. However, if the matrix A is tridiagonal, then the only values of k which may contribute to the sum are $(i-1)$, i and $(i+1)$ (since A_{ik} is zero for all other values of k). So, for A tridiagonal,

$$P_i \equiv A_{i(i-1)}V_{i-1} + A_{ii}V_i + A_{i(i+1)}V_{i+1}, \quad i \neq 1, n.$$

Thus, the computation of one element of P requires at most only 3 multiplications and 2 additions.

However, the execution time of an algorithm is determined by more than just how many arithmetic operations are to be performed: there is also the question of how well operations map onto the target architecture. For example, array index expressions such as V_{i-1} can introduce overheads for 'shifting' arrays on some architectures. Such overheads are usually small for tridiagonal matrices, due to the regularity of the index expressions. However, the overheads for unstructured matrices can be large.

4.2 Example: Matrix Product

Consider the expression A*B, which denotes the product of two real matrices A and B, which are initially marked as being of shape $n \times n$. The initial stages of a derivation convert this expression (and all the definitions of library functions) into λ-calculus form, and then unfold the definitions, to give the following expression for the product:

```
generate([n,n],
  λ[i,j]·reduce([n],λ[k]·A[i,k]*B[k,j],+,0))
```

Now assume that A is marked as being tridiagonal. Operations involving A are converted into equivalent operations on the $n \times 3$ primary store for A, which we designate, Ap.

```
generate([n,n],λ[i,j]·
    if (i=1) then 0 else Ap[i,1]*B[i-1,j]
  + Ap[i,2]*B[i,j]
  + if (i=n) then 0 else Ap[i,3]*B[i+1,j])
```

Note that:

- the reduction of n values has been converted into a *pair* of additions;
- the conditional expressions allow for there being only two non-zero elements in the first and last rows of a tridiagonal matrix;
- accessing elements on the diagonal and sub- and super-diagonals has been converted into accessing elements in the appropriate column of the primary store.

From this form, the latter stages of the derivation produce imperative forms. For example, the Fortran77 sub-derivation produces the implementation shown in Figure 5(a) while the combination of the Array and DAP sub-derivations produces the implementation shown in Figure 5(b):

Fortran77 implementation:

The final loop uses column-updating (i-j order) rather than row-updating (j-i order) on the product array P; i-j ordering arose from a straightforward translation of generate expressions, in which the indices happen to be listed in that order. In this particular example, the two orderings turn out to be equivalent for the CRAY because the CRAY compiler exchanges the loops in the i-j case.

DAP implementation:

- A(,j) (with the row index position blank) denotes column j of matrix A;
- shnp(B) denotes matrix B shifted one place to the north, with zeros being introduced along the south edge (row 1 being the north edge of the matrix);
- shsp(B) is B shifted one place to the south.

It is important to emphasize that the sequential and Array/DAP sub-derivations are independent of the tridiagonal sub-derivation; they are the same sub-derivations that would be applied to produce implementations for dense matrices.

In this paper, we discuss the derivation of the sparse form. The derivation of the imperative implementations is discussed elsewhere.

4.3 Tridiagonal Sub-derivation

The tridiagonal sub-derivation converts expressions in which matrices may have been marked as being tridiagonal into equivalent expressions involving the primary stores of the matrices. The sub-derivation proceeds in three main steps:

Marking: the zero and non-zero elements of tridiagonal matrices are distinguished by introducing a conditional expression that explicitly checks whether an index falls within the tridiagonal region.

```
  real Ap(n,3),B(n,n),P(n,n)
  do 1 j=1,n,1
  P(1,j) = Ap(1,2)*B(1,j)+Ap(1,3)*B(2,j)
1 continue
  do 2 j=1,n,1
  P(n,j) = Ap(n,1)*B(n-1,j)+Ap(n,2)*B(n,j)
2 continue
  do 3 i=2,n-1,1
  do 3 j=1,n,1
  P(i,j) = Ap(i,1)*B(i-1,j)+Ap(i,2)*B(i,j)+Ap(i,3)*B(i+1,j)
3 continue
```

(a) Fortran77

```
matc(Ap( ,1),n)*shnp(B)
+matc(Ap( ,2),n)*B
+matc(Ap( ,3),n)*shsp(B)
```

(b) DAP

Figure 5: Imperative implementations of the product of matrices A and B, where A is tridiagonal

Simplification: algebraic properties of the λ-calculus and various data types (primarily real numbers and arrays) are used to simplify expressions. (e.g. $x + 0 \rightarrow x$.)

Mapping: operations on tridiagonal matrices are converted into operations on the corresponding primary stores. (Some further simplification is possible after this mapping.)

Of the three steps, only the final, mapping step is heavily dependent on the nature of the sparsity. The marking step and the simplification step are mostly independent of the type of sparsity.

These steps are discussed further below, and are illustrated using matrix product. The tridiagonal sub-derivation is applied to the expression for matrix product produced by the initial stages of the derivation, *viz.*:

```
generate([n,n],
 λ[i,j]·reduce([n],λ[k]·A[i,k]*B[k,j],+,0))
```

It is assumed that matrix A has been marked as being tridiagonal (matrix B is assumed to be dense).

4.3.1 Marking Step

An array element expression such as A[i,j], where A is sparse, is zero if the index [i,j] falls outside the non-zero region (by definition). Thus, the expression A[i,j] is equivalent to the conditional expression

```
if ([i,j] ∈ tridiagonal(n)) then A[i,j] else 0
```

The advantages of the conditional form are:

- The zeros are explicitly distinguished from the non-zeros. Explicitness generally facilitates program transformation (by providing some entity in a program which can be directly manipulated by transformations).
- The zeros are explicitly denoted by the number 0 (rather than by expressions of the form A[i,j]).
- All remaining array element expressions are known to refer to non-zeros.

Thus, the first step of the tridiagonal sub-derivation is to convert all element expressions for tridiagonal matrices into conditional expressions.

```
.sd.
 A[i,j]
 ==>
 if ([i,j] ∈ tridiagonal(n)) then A[i,j] else 0
  where A is marked as being tridiagonal(n)
.sc.
```

The expression for matrix product is transformed into

```
generate([n,n],
 λ[i,j]·reduce([n],λ[k]·A[i,k]*B[k,j],+,0))
==>
generate([n,n],
 λ[i,j]·reduce([n],
  λ[k]·(if ([i,k] ∈ tridiagonal(n))
   then A[i,k]
   else 0)*B[k,j],
  +,0))
```

4.3.2 Simplification Step

The simplification of expressions is based mainly upon properties of real numbers such as

$$x + 0 \equiv x \quad \text{and} \quad x * 0 \equiv 0 \ .$$

These lead immediately to the transformations:[5]

```
.sd. x+0 ==> x .sc.     and     .sd. x*0 ==> 0 .sc. .
```

Further simplifications are more specific to sparse sub-derivations. In this section, the simplification of matrix product is illustrated.

The expression produced by the marking step is:

[5] All function applications are in prefix form by this stage of the derivation but we use infix operator notation to enhance the readability of examples.

```
generate([n,n],
 λ[i,j]·reduce([n],
  λ[k]·(if ([i,k] ∈ tridiagonal(n))
   then A[i,k]
   else 0)*B[k,j],
  +,0))
```

This is simplified as follows:

- The application of * is propagated into the conditional expression by the transformation

```
.sd.
 f(if ([i,j] ∈ tridiagonal(n)) then T else F)
 ==>
 if ([i,j] ∈ tridiagonal(n)) then f(T) else f(F)
.sc.
```

to give

```
==> generate([n,n],
 λ[i,j]·reduce([n],
  λ[k]·if ([i,k] ∈ tridiagonal(n))
   then A[i,k]*B[k,j]
   else 0*B[k,j],
  +,0))
```

- The expression, 0*B[k,j], is simplified:

```
generate([n,n],
 λ[i,j]·reduce([n],
  λ[k]·if ([i,k] ∈ tridiagonal(n))
   then A[i,k]*B[k,j]
   else 0,
  +,0))
```

- For a particular value of i, the predicate ([i,k] ∈ tridiagonal(n)) is satisfied for, at most, three values of k, *viz.* $(i-1)$, i and $(i+1)$. Since the generating function returns the identity (0) of the reducing function (+) for all other values of k, the reduction can be replaced by a pair of additions (adding the values for $k = i-1, i, i+1$):

```
generate([n,n],
 λ[i,j]·(if (i=1) then 0 else A[i,i-1]*B[i-1,j])
 + A[i,i]*B[i,j]
 + (if (i=n) then 0 else A[i,i+1]*B[i+1,j]))
```

4.3.3 Mapping Step

The marking and simplification steps have restricted element expressions to involve only elements in the non-zero region, and have optimized operations by restricting them to the tridiagonal region, but the tridiagonal matrices are still sparse (i.e. they are still $n \times n$ matrices with most of their elements zero). The final step of the sub-derivation maps the non-zero region onto the primary store, a dense matrix with 3 columns.

The conversion to compact form is based on the following two mappings:

Shapes: $tridiagonal(n) \rightarrow [n, 3]$;

Indices: $[i, j] \rightarrow [i, j-i+2]$ and the inverse $[i', j'] \rightarrow [i', i'+j'-2]$.

For transformations, only element access and reductions need be considered (generations need not be considered since it is assumed that tridiagonal matrices are not constructed; the transformation of generations is, however, similar to the transformation of the generating functions of reductions).

Mapping element accesses is a straightforward application of the above identities:

```
.sd.
 A[i,j] ==> Ap[i,j-i+2]
  where A is an order n tridiagonal matrix
  and where Ap is the n × 3 primary store for A
.sc.
```

The mapping of reductions is more complex. A reduction is evaluated by applying a reducing function to a set of values that is produced by applying a generating function to the indices in some set. Suppose a reduction is over a tridiagonal region. The mapping step has to convert this region into a rectangular region. The generating function must be altered so that, when applied to the rectangular region, it produces the same values as it originally produced when applied to the tridiagonal region. That is, if an index $[i', j']$ in the set $[n, 3]$ corresponds to index $[i, j]$ in the set $tridiagonal(n)$, then it is required that $g'(i', j') \equiv g(i, j)$, where g' is the altered generating function and g is the original generating function.

The altered generating function can be formed from the original generating function by applying the inverse index mapping. However, the invalid locations ($[1, 1]$ and $[n, 3]$) in the $[n, 3]$ index set must be excluded from the reduction; this is achieved by having the altered generating function return an identity element of the reducing function for these indices. Later transformations may be able to incorporate the invalid indices if the generating function returns the identity element when applied to the invalid indices.

Thus, the transformation for reductions is:

```
.sd.
 reduce(tridiagonal(n),λ[i,j]·g,r,init)
 ==>
 reduce([n,3],
  λ[i′,j′]·if (valid([i′,j′]))
   then λi,j·g (i,j-i+2)
   else r0,
  r,init)
.sc.
```

where valid checks if an index is not one of the invalid locations, and r0 is an identity of r.

Figure 6 traces the application of the above transformations to the matrix product expression. (Some further transformations are applied to simplify arithmetic expressions in indices.) The final expression is the desired expression for the tridiagonal-optimized matrix product. As pointed out earlier, this expression is processed by other sub-derivations to produce imperative implementations (see Figure 5).

```
generate([n,n],
 λ[i,j]·(if (i=1) then 0 else A[i,i-1]*B[i-1,j])
 + A[i,i]*B[i,j]
 + (if (i=n) then 0 else A[i,i+1]*B[i+1,j]))

==>  Element access transformation applied to A
generate([n,n],
  λ[i,j]·(if (i=1) then 0 else Ap[i,(i-1)-i+2]*B[i-1,j])
 + Ap[i,i-i+2]*B[i,j]
 + (if (i=n) then 0 else Ap[i,(i+1)-i+2]*B[i+1,j]))

==>  Arithmetic simplification of index expressions
generate([n,n],
 λ[i,j]·(if (i=1) then 0 else Ap[i,1]*B[i-1,j])
 + Ap[i,2]*B[i,j]
 + (if (i=n) then 0 else Ap[i,3]*B[i+1,j]))
```

Figure 6: Mapping onto primary store, Ap, for matrix product A*B where A is tridiagonal

$$\underbrace{\begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 2 & 3 & 4 & 0 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 0 & 6 & 7 & 8 \\ 0 & 0 & 0 & 8 & 9 \end{bmatrix}}_{\text{The } \textit{sparse matrix}} \longrightarrow \underbrace{\begin{bmatrix} - & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix}}_{\text{The } \textit{primary store}}$$

Figure 7: Storage for a symmetric tridiagonal matrix

5 Symmetric Tridiagonal Matrices

In this section, the tridiagonal sub-derivation is modified for *symmetric* tridiagonal matrices. A symmetric matrix possesses symmetry about the main diagonal: i.e. $A_{ij} = A_{ji}$. For a tridiagonal matrix, symmetry implies that the elements in the sub-diagonal are the same as the elements in the super-diagonal. Storage requirements can thus be reduced by storing only the sub-diagonal and main diagonal, as illustrated in Figure 7.

This storage scheme is characterized by the mappings:

Shape: $symtridiag(n) \rightarrow [n, 2]$;

Indices: $[i, j] \rightarrow \begin{cases} [i, j - i + 2] & \text{for } j \leq i \\ [i + 1, 2 - (j - i)] & \text{for } j > i. \end{cases}$

Since this mapping is not 1-1, the 'inverse' is more complex than for the tridiagonal case; it is discussed below in the context of converting reductions.

5.1 The Symmetric Tridiagonal Sub-derivation

The symmetric tridiagonal sub-derivation is similar to the tridiagonal sub-derivation. It proceeds through three main steps: marking, simplification and mapping.

- As in the tridiagonal sub-derivation, the marking step distinguishes the non-zero region from the zero region. However, the symmetric tridiagonal marking step also restricts all element accesses to the lower triangle.
- The simplification step is the same as for the tridiagonal sub-derivation.
- The mapping step is similar in form to the tridiagonal mapping step, though of course it differs in details.

5.1.1 Marking Step

The marking step introduces conditional expressions to distinguish elements in the zero region from those in the non-zero region.

```
.sd.
 A[i,j]
 ==>
 if ([i,j] ∈ symtridiag(n)) then A[i,j] else 0
  where A is marked as being symmetric tridiagonal
.sc.
```

It then introduces conditional expressions to restrict all element accesses to the lower triangle.

```
.sd.
 A[i,j] ==> if (j>i) then A[j,i] else A[i,j]
  where A is marked as being symmetric
.sc.
```

As before, each step of the derivation is illustrated using matrix product. The expression for matrix product after conversion to λ-calculus and unfolding is:

```
generate([n,n],
 λ[i,j]·reduce([n],λ[k]·A[i,k]*B[k,j],+,0))
```

After the marking step, the expression is

```
generate([n,n],
 λ[i,j]·reduce([n],
  λ[k]·(if ([i,k] ∈ symtridiag(n))
    then if (k>i) then A[k,i] else A[i,k]
    else 0
    )*B[k,j],
  +,0))
```

5.1.2 Simplification Step

The rules for simplification are similar to those for the tridiagonal sub-derivation; the simplification of matrix product is shown in Figure 8. The conditional expressions, introduced in the previous step to restrict element access to the lower triangle, are simplified using the identities: $i - 1 > i \equiv false$, $i > i \equiv false$, and $i + 1 > i \equiv true$.

```
generate([n,n],λ[i,j]·reduce([n],λ[k]·
 (if ([i,k] ∈ symtridiag(n))
 then if (k>i) then A[k,i] else A[i,k]
 else 0 )*B[k,j],
 +,0))

==>  Propagation through conditionals and multiplication by 0
generate([n,n],λ[i,j]·reduce([n],λ[k]·
 if ([i,k] ∈ symtridiag(n))
 then if (k>i) then A[k,i]*B[k,j] else A[i,k]*B[k,j]
 else 0,
 +,0))

==>  Restriction of the reduction to non-zeros: k→(i-1), i, (i+1)
generate([n,n],λ[i,j]·
   (if (i=1) then 0 else if (i-1>i) then A[i-1,i]*B[i-1,j] else A[i,i-1]*B[i-1,j])
 + if (i>i) then A[i,i]*B[i,j] else A[i,i]*B[i,j]
 + (if (i=n) then 0 else if (i+1>i) then A[i+1,i]*B[i+1,j] else A[i,i+1]*B[i+1,j])
 )

==>  Simplification
generate([n,n],λ[i,j]·
   (if (i=1) then 0 else A[i,i-1]*B[i-1,j])
 + A[i,i]*B[i,j]
 + (if (i=n) then 0 else A[i+1,i]*B[i+1,j])
 )
```

Figure 8: Simplification of matrix product for symmetric tridiagonal matrix

5.1.3 Mapping Step

As in the tridiagonal sub-derivation, the mapping step of the symmetric tridiagonal sub-derivation must convert element accesses and reductions.

The conversion of element access is based on the index mapping presented earlier:

$$[i,j]\rightarrow \begin{cases} [i, j-i+2] & \text{for } j \le i \\ [i+1, 2-(j-i)] & \text{for } j > i. \end{cases}$$

Because the marking stage of the sub-derivation restricts all element accesses to the lower triangle, the condition $j \le i$ always holds, so the transformation is a straightforward application of the corresponding mapping:

```
.sd.
 A[i,j]
 ==>
 Ap[i,j-i+2]
  where A is marked as being symmetric tridiagonal
.sc.
```

A reduction over the order n symmetric tridiagonal region is converted into a reduction over the rectangular shape $n \times 2$. However, each index in the first column of this shape corresponds to *two* indices in the symmetric tridiagonal shape: one in the lower triangle and one, corresponding by reflection in the main diagonal, in the upper triangle (see Figure 9).

$$j' = 1 : [i',j'] \rightarrow \begin{cases} [i', i'+j'-2] & \text{lower triangle} \\ [i'+j'-2, i'] & \text{upper triangle} \end{cases}$$

Thus, a reduction over the symmetric tridiagonal region is converted into three reductions: two over the first column of the primary store (for which j' is 1), and one over the second column of the primary store (for which j' is 2). One of the reductions over the first column uses the second of the two mappings above ($[i',j'] \rightarrow [i'+j'-2, i']$); the other two reductions use the first mapping ($[i',j'] \rightarrow [i', i'+j'-2]$). See Figure 10.

The matrix product example requires only the element access transformation: see Figure 11. The final expression is the form for matrix product produced by the symmetric tridiagonal sub-derivation. Further sub-derivations convert this expression into imperative form. For example, Figure 12 shows sequential and DAP implementations.

6 Band Matrices

An obvious generalization of tridiagonal matrices are matrices having arbitrary numbers of sub- and super-diagonals; an

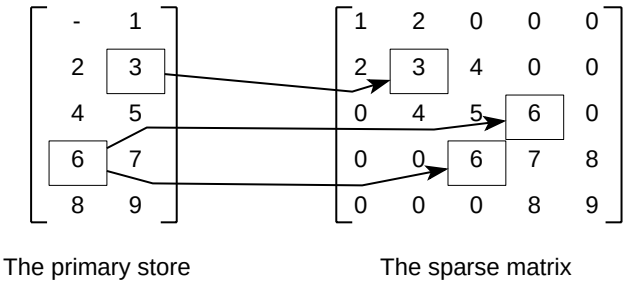

Figure 9: Mapping from primary store onto sparse matrix for a symmetric tridiagonal matrix

```
.sd.
 reduce(symtridiag(n),λ[i,j]·g,r,init)
 ==>
 r(init,
   reduce([n],λ[i′]·λj′·(if (i′=1) then r0 else λi,j·g (i′,i′+j′-2)) (1),r,r0),
   r(reduce([n],λ[i′]·λj′·λi,j·g (i′,i′+j′-2) (2),r,r0),
     reduce([n],λ[i′]·λj′·(if (i′=1) then r0 else λi,j·g (i′+j′-2,i′)) (1),r,r0)
   )
 )
  where r0 is an identity of r
.sc.
```

Figure 10: Transformation for reduction over symmetric tridiagonal region

```
generate([n,n],
  λ[i,j]·(if (i=1) then 0 else A[i,i-1]*B[i-1,j])
  + A[i,i]*B[i,j]
  + (if (i=n) then 0 else A[i+1,i]*B[i+1,j])
  )

==>  Element access mapping
generate([n,n],
   λ[i,j]·(if (i=1) then 0 else Ap[i,(i-1)-i+2]*B[i-1,j])
  + Ap[i,i-i+2]*B[i,j]
  + (if (i=n) then 0 else Ap[i+1,i-(i+1)+2]*B[i+1,j])
  )

==>  Arithmetic simplification
generate([n,n],
  λ[i,j]·(if (i=1) then 0 else Ap[i,1]*B[i-1,j])
  + Ap[i,2]*B[i,j]
  + (if (i=n) then 0 else Ap[i+1,1]*B[i+1,j])
  )
```

Figure 11: The mapping step applied to matrix product for symmetric tridiagonal matrix

```
real Ap(n,2),B(n,n),P(n,n)
do 1 j = 1,n,1
P(n,j) = Ap(n,1)*B(n- 1,j)+Ap(n,2)*B(n,j)
1 continue
do 2 j = 1,n,1
P(1,j) = Ap(1,2)*B(1,j)+Ap(2,1)*B(2,j)
2 continue
do 3 i = 2,n- 1,1
do 3 j = 1,n,1
P(i,j) = Ap(i,1)*B(i-1,j)+Ap(i,2)*B(i,j)+Ap(i+1,1)*B(i+1,j)
3 continue
```

(a) Fortran77

```
real Ap(*n,*2),B(*n,*n),P(*n,*n)
P = matc(Ap( ,1),n)*shsp(B)
 +matc(Ap( ,2),n)*B
 +matc(shlp(Ap( ,1),1 ),n)*shnp(B)
```

(b) DAP

Figure 12: Implementations of matrix product A*B for A symmetric tridiagonal

order n matrix with l sub-diagonals and u super-diagonals may be denoted $band(n, l, u)$.[6]

One storage scheme for such a matrix is to map each diagonal onto a column of a matrix, of width $l + u + 1$:

- column $l + 1$ stores the main diagonal;
- columns to the left store the sub-diagonals in order, with column l storing the first, column $(l - 1)$ storing the second, etc.;
- columns to the right store the super-diagonals in order, with column $(l + 2)$ storing the first, column $(l + 3)$ storing the second, etc.

Figure 13 shows a $band(9, 3, 2)$ matrix and its primary store.

The mappings defining this storage scheme are:

Shape: $band(n, l, u) \rightarrow [n, l + u + 1]$;

Indices : $[i, j] \rightarrow [i, j - i + l + 1]$ and the inverse $[i', j'] \rightarrow [i', i' + j' - (l + 1)]$.

The transformations discussed previously for tridiagonal matrices can be extended to use the above mappings to optimize specifications to operate on banded matrices. The optimized form of matrix product is shown in Figure 14.

7 Row-wise Sparse Matrices

'Row-wise sparse' is the term used here for a matrix which has a fixed number of non-zeros in each row [9] and where the locations of the non-zeros do not correspond to any pattern (the positions are not known until execution time and are specified using a lookup table).[7]

An $n \times m$ row-wise sparse matrix A having w non-zeros per row (w is called the 'width' of the matrix) is denoted $rowwise([n, m], w)$. Such a matrix can be compactly represented using an $n \times w$ primary store A_p — containing the non-zero elements — and an $n \times w$ secondary store A_s — containing the column indices for the non-zero elements.

Figure 15 illustrates a row-wise sparse matrix. Note that the second row contains one non-zero fewer than the other rows; one of that row's zero elements is included in the compact stores to give a uniform width throughout the matrix.

The relationship between the sparse matrix A and the compact stores is formally given by the mappings:

Shapes: $rowwise([n, m], w) \rightarrow [n, w]$;

Indices $[i, j] \rightarrow [i, search([i, j])]$
and the inverse $[i, j] \rightarrow [i, lookup([i, j])]$.

Given a sparse index (i, j), $search$ returns the column index of the element in row i of the primary store that contains the value for element (i, j). It is assumed that locating this element requires a scan of the corresponding row of the secondary store and is thus a relatively expensive operation.

Given a primary index, $lookup$ returns the corresponding sparse index. This can be implemented as a simple array index of the secondary store, and so is a relatively inexpensive operation.

The two functions are related by the identity

$$search([i, lookup([i, j])]) \equiv j$$

which states that the sparse index corresponding to the primary index which corresponds to a sparse index j, is j.

7.1 Row-wise Sub-derivation

The sub-derivation to optimize expressions containing operations on row-wise sparse matrices is similar in structure to that for tridiagonal matrices; specifically, the row-wise sub-derivation has the same three steps: marking, simplification and mapping. These steps are discussed below. Matrix product is used as an example; the expression for matrix-product before application of the row-wise sub-derivation is:

```
generate([n,n],
  λ[i,j]·reduce([n],λ[k]·A[i,k]*B[k,j],+,0))
```

[6] The extension of the symmetric tridiagonal sub-derivation for symmetric band matrices has not yet been attempted, though no serious technical problems are anticipated. It is likely that the transformations relating to symmetry can be separated into a distinct sub-derivation, which can then be combined with the sub-derivation discussed in this section.

[7] A band matrix could be represented as a row-wise matrix, but it is not desirable to do so as advantage could not be taken of the regular pattern characterizing the band matrix.

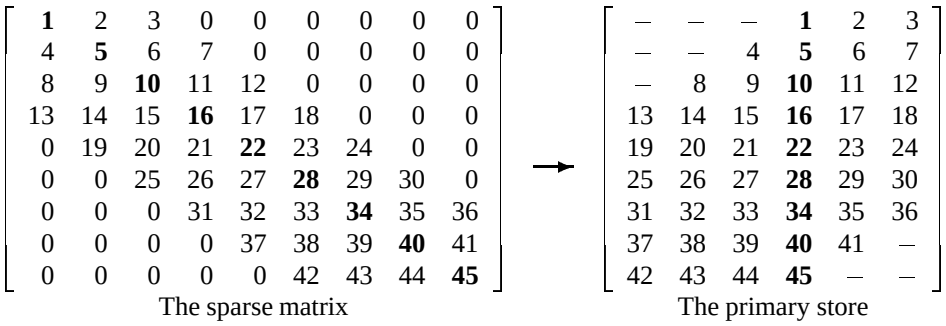

Figure 13: A $band(9,3,2)$ sparse matrix; the elements on the main diagonal are highlighted

```
generate([n,n],λ[i,j]·
  if (i<l) then reduce([(1,i+u,1)],λ[k]· Ap[i,k-i+l+1]*B[k,j],+,0)
  else if (i>n-u) then reduce([(i-l,n,1)],λ[k]· Ap[i,k-i+l+1]*B[k,j],+,0)
  else reduce([(i-l,i+u,1)],λ[k]· Ap[i,k-i+l+1]*B[k,j],+,0))
```

Figure 14: Matrix product $A * B$; A is a band matrix having l lower and u upper diagonals

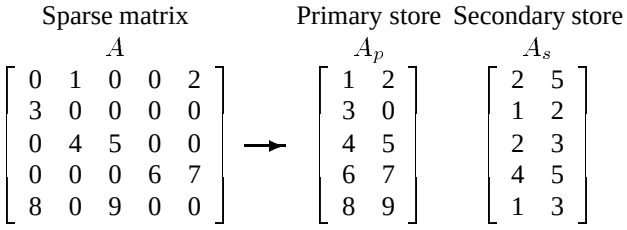

Figure 15: Storing a row-wise sparse matrix

```
generate([n,n],λ[i,j]·reduce(row(i,rowwise([n,n],w)),λ[k]·A[i,k]*B[k,j],+,0))

==> mapping transformations
generate([n,n],λ[i,j]·reduce([w],λ[k′]·λ[k]·Ap[i,search([i,k])]*B[k,j] (lookup([i,k′])),+,0))

==> β-reduce the binding for k
generate([n,n],λ[i,j]·reduce([w],λ[k′]·Ap[i,search([i,lookup([i,k′])])]*B[lookup([i,k′]),j],+,0))

==> apply search-lookup identity
generate([n,n],λ[i,j]·reduce([w],λ[k′]·Ap[i,k′]*B[lookup([i,k′]),j],+,0))
```

Figure 16: Mapping step for row-wise sparse applied to matrix product

Matrix A is assumed to have been specified as being row-wise sparse, of order $n \times n$ and width w.

7.1.1 Marking Step

Conditional expressions are introduced in place of array indexing applied to row-wise sparse matrices.

```
==> generate([n,n],
 λ[i,j]·reduce([n],
  λ[k]·(if ([i,k] ∈ rowwise([n,n],w))
   then A[i,k]
   else 0)*B[k,j],
  +,0))
```

7.1.2 Simplification Step

The application of real multiplication is propagated through the conditional expression, producing a multiplication by zero, which is simplified:

```
==> generate([n,n],
 λ[i,j]·reduce([n],
  λ[k]·if ([i,k] ∈ rowwise([n,n],w))
   then A[i,k]*B[k,j]
   else 0,
  +,0))
```

As with the tridiagonal sub-derivation, the reduction over k contains a conditional expression whose predicate is satisfied for only a distinct subset of the values assumed by k — those values for which the index [i,k] corresponds to a non-zero. The notation row(i,rowwise([n,n],w)) is used to indicate this subset. The expression for matrix product then becomes:

```
==> generate([n,n],
 λ[i,j]·reduce(row(i,rowwise([n,n],w)),
  λ[k]·A[i,k]*B[k,j],+,0))
```

7.1.3 Mapping Step

The mapping step converts operations on the $n \times n$ sparse matrix A into equivalent operations on the compact stores Ap and As. The general identities underlying this conversion were given above; an additional identity is required to specify the mapping for the row(i,rowwise(...)) notation:

$$row(i, rowwise([n,n],w)) \rightarrow [w]$$

that is, the w column indices corresponding to the non-zeros in a given row are mapped onto the indices $\{k : 1 \leq k \leq w\}$.

Applying these identities to matrix product produces an expression that would be very inefficient if implemented directly, since it requires searching the secondary store for indices. However, the expression can be simplified using the identity

$$search([i, lookup([i,j])]) \equiv j$$

as shown in Figure 16.

The presence of lookup in the resulting expression is not as potentially detrimental to performance as the presence of search would be, since lookup is implemented as a simple array index on the secondary store.

```
==> generate([n,n],
 λ[i,j]·reduce([w],λ[k′]·Ap[i,k′]*B[As[i,k′],j],+,0))
```

Further sub-derivations are applied to the expression produced by the mapping step to derive Fortran77 implementations suitable for execution on sequential and vector processors (see Figure 17 for the basic implementation). No implementation is produced for the DAP array processor, because the indirect array indexing (B[As[i,k′],j]) is virtually impossible to implement efficiently on the DAP hardware. Efficient implementation may, however, be possible on other array processors having more sophisticated communication mechanisms.

8 Results

The preceding sections of this paper have discussed sub-derivations that optimize expressions for various types of sparse matrices. Further sub-derivations are then used to produce Fortran implementations suitable for sequential, vector and array processors, as discussed in Section 3.3.

In this section, the derivations are assessed by comparing the execution performance of derived implementations of two non-trivial algorithms with the performance of manually constructed implementations.

```
      do 20 i=1,n,1
      do 20 j=1,n,1
      res(i,j)=0.0
      do 10 k=1,w,1
      res(i,j)=res(i,j)+Ap[i,k]*B[As[i,k],j]
   10 continue
   20 continue
```

Figure 17: Matrix product A*B; A is row-wise sparse with width w

8.1 Two Examples

The examples given above occur in many applications of numerical mathematics, and are simple enough to serve as illustrations for the derivation process. However, the advantages of the derivational approach to implementation is more evident when complex algorithms are considered.

Here, two algorithms are considered: POT [10], an algorithm for computing the eigensystem of real symmetric matrices; and a Conjugate Gradient algorithm (CG) [11], for computing the solution of a set of simultaneous equations.

We present both the mathematical formulation and the functional specification of POT to allow comparison of the two; we present only the functional specification of CG, partly for brevity, partly because the functional specification has assumed the rôle of being *the* definition of the algorithm (the name "Conjugate Gradient" refers more to a family of algorithms than to a single algorithm — the functional specification is the precise definition of the particular member of the family considered in this paper).

Example 8.1-a: *POT*

The mathematical definition of POT is shown in Figure 18 and the functional specification in Figure 19.

This example exhibits several features of the specification language not previously mentioned:

- The notation

  let *definitions* in *expression* ... end

  denotes a *local expression*. Local expressions are used to introduce definitions that are valid only within the local expression itself. For example, POTstep defines the four values V, B, T and R′. The value of a local expression is the value of the *body expression* occurring after the in keyword: (A,orthonormalise(R′)).

- The expression (A,orthonormalise(R′)) is a pair, or a 2-tuple, of real matrices. Components of tuples are extracted using the #n prefix operators, where n is an integer. For example, #1(A,orthonormalise(R′)) is A.

- The function iterate repeatedly applies a function until a specified condition is met. Here, the function that is repeatedly applied is POTstep; the value to which it is initially applied is (A, UnitMatrix(shape(A))); and the terminating condition is defined by the function IsAccurate (which determines whether or not a given approximation to the eigensystem is sufficiently accurate). Thus, this application of iterate realizes the repetition required by POT.

Example 8.1-b: *Conjugate Gradient*

The Conjugate Gradient algorithm (CG) uses an iterative process to compute the vector x of order n satisfying the equation $Ax = b$ where A is a positive definite, symmetric matrix of order $n \times n$ and b is a vector of order n. The functional specification of CG is shown in Figure 20

- The algorithm is based upon manipulation of a set of vectors x,r,p and q (x being the current approximation to the solution); the type cgstate is defined to represent the set of vectors, as a 4-tuple of real vectors. Instances of the cgstate type are constructed using the function cgstate.

- The function cgiters takes A and b as arguments and returns a cgstate whose first component is the approximate solution.

- The algorithm is iterative, thus cgiters is an application of the iterate library function.
  - The first argument for iterate is a function defining the computation that is to be repeated: cgiter.
  - The second argument is a value (an instance of cgstate) with which to begin the repetition.
  - The third argument, isok, is a function which determines when the repetition is to cease (i.e. when the approximation to the solution is sufficiently accurate).

- The function defining the repeated computation, cgiter, takes one argument of type cgstate and returns a value of the same type. In the specification, pattern matching is used to bind the names x,r,p and q to the four components of the cgstate argument.

- The body of cgiter computes the next set of vectors as local values x′, r′, p′ and q′ and returns these values as an instance of cgstate.

- For brevity, the computation of the initial values x0, r0, p0 and q0 is not shown.

- The bulk of the computational costs are incurred by the two matrix-vector products, in the computation of atq and q′.

8.2 Assessment of Derived Implementations

The effectiveness of derivations can be assessed by comparing the execution times of derived implementations with those of manually constructed implementations.

- For POT, an independent implementation is available for symmetric tridiagonal matrices [10].

The POT algorithm for the computation of the eigensystem of a real symmetric matrix A, of order N, consists essentially of the following iterative cycle. Given an $N * N$ matrix R of orthonormal approximations to the eigenvectors of A:

1. Form a transformed matrix B, of order N, using the sequence of algebraic operations:
$$V := A * R, B := R^T * V.$$

2. Construct a transformation matrix T, of order N, whose columns form an approximation to the set of right eigenvectors of the symmetric matrix B. This construction is described in [8].

3. Compute a new set of eigenvector approximations R, which is represented in matrix form by the algebraic matrix product
$$R := V * T.$$

4. Orthonormalise the set of vectors R.

The cycle ceases whenever the off-diagonal components of B are zero to a predetermined precision. The diagonal of B then yields the required eigenvalues and the columns of the final matrix R yield the required eigenvectors.

Figure 18: Definition of POT [10]

```
fun POTstep(A:real matrix,R:real matrix):(real matrix*real matrix)
 = let
    val V:real matrix = A*R
    val B:real matrix = transpose(R)*V
    val T:real matrix = transform(B)
    val R′:real matrix = V*T
   in
    (A,orthonormalise(R′))
   end

fun POT(A:real matrix):(real matrix*real matrix)
 = iterate(POTstep,(A,UnitMatrix(shape(A))),IsAccurate)
```

Figure 19: Functional specification of POT

```
type cgstate = real vector*real vector*real vector*real vector;
fun cgiters(a:real matrix,b:real vector):cgstate
 = let
  (* Terminating condition.*)
  fun isok((x,r,p,q):cgstate):bool
   = innerproduct(r,r)<epsilon;

  (* One iteration.*)
  fun cgiter((x,r,p,q):cgstate):cgstate
   = let
    val rr:real = innerproduct(r,r);
    val alpha:real = rr/innerproduct(q,q);
    val x′:real vector = x+p*alpha;
    val atq:real vector = transpose(a)*q;
    val r′:real vector = r-atq*alpha;
    val beta:real = innerproduct(r′,r′)/rr;
    val p′:real vector = r′+p*beta;
    val q′:real vector = a*r′+q*beta
   in
    cgstate(x′,r′,p′,q′)
   end
 in
  iterate(cgiter,cgstate(x0,r0,p0,q0),isok)
 end
```

Figure 20: SML specification of *bi-conjugate* gradient

- For CG, an implementation can be constructed that makes use of the BLAS SGBMV routine [12]. This routine performs the operation

  $$y := \alpha Ax + \beta y$$

  where A is a sparse matrix with an arbitrary number of contiguous sub- and super-diagonals, x and y are dense vectors and α and β are scalars. There are two implementations of SGBMV available on the CRAY vector computer, one provided by CRAY (referred to as the COS implementation) and one provided by NAG. It is assumed that these implementations are highly efficient.

  This routine can be used to perform the computation of r' and q':

  ```
  val atq:real vector = transpose(a)*q;
  val r′:real vector = r-atq*alpha;
  . . .
  val q′:real vector = a*r′+q*beta
  ```

  In a dense implementation of CG, the computations of r' and q' dominate the computational costs. However, when matrix A is sparse, these computations become relatively less expensive. Nevertheless, they still contribute significantly to the overall computational cost:

  – In practical applications of CG, the proportion of the total computational cost that arises from the computation of r' and q' is smallest when A is tridiagonal. Then, each matrix-vector product is equivalent, in terms of computational cost, to three vector elementwise operations (such as the addition of two vectors or the scaling of a vector); thus, the computation of r' and the computation of q' each has the same cost as 5 vector elementwise operations.

  – The total cost of all other computations in the main iterative section of the algorithm is equivalent to 7 vector elementwise operations (taking the cost of computing an inner-product to be the same as the cost of an elementwise operation), regardless of whether or not A is sparse.

  – Thus, the computations of r' and q' account for at least 60% (approximately) of the total computational costs.

So, the execution performance of CG provides a fairly sensitive measure of the quality of the implementation of the computations of r' and q'. A comparison of the execution performance of two implementations of CG — one a derived implementation and the other an implementation in which the computations of r' and q' are performed using a BLAS library routine — thus provides a measure for the overall quality of the derived implementation.

8.2.1 POT on DAP

Figure 24 (in the appendix) shows the derived implementation of POT (including the orthogonalization step) for a symmetric tridiagonal matrix. Figure 21 compares the execution times for one iteration[8] of the derived and manually constructed implementations. The execution times for the two are virtually identical, with the derived implementation being marginally slower. Analysis of the derived implementation reveals a few potential optimizations that could be exploited by transformations.

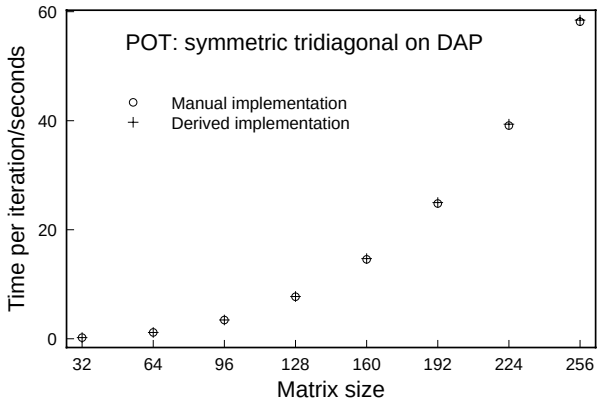

Figure 21: Execution times for one iteration of derived and manually constructed DAP implementations of POT for symmetric tridiagonal matrix.

8.2.2 Conjugate Gradient on CRAY

The derived implementation of CG for band matrices, optimized for the CRAY vector processor, is shown in Figure 25 (in the appendix). This implementation was compared over a range of band widths with one in which the computation of r' and q' was performed using the CRAY implementation of the SGBMV BLAS routine. Figure 22 shows typical execution times for the derived, COS BLAS and NAG BLAS implementations.

If the band parameters are specified as 'compile-time' constants, then the loops terminating at lines 170 and 270 (in the derived implementation) can be unrolled by the compiler.[9] Loop unrolling is not feasible for the precompiled library subroutines: these subroutines are not amenable to manipulation since their source codes are not available. Inevitably this means that the library routines are at a disadvantage in any performance comparison: the subroutine libraries have been designed to operate with matrices having *any* band size and this generality precludes optimizations for *particular* band sizes.

However, this limitation of monolithic subroutine libraries is one of the points in favour of the derivational approach! If a programmer chooses, as many do, to simplify the implementation of an algorithm by employing subroutine libraries,

[8]One iteration corresponds to the computation of one approximation to the eigensystem. The time for successive iterations does not change.

[9]Unrolling of loops could be performed by transformations; it was convenient in this case to leave unrolling to the compiler.

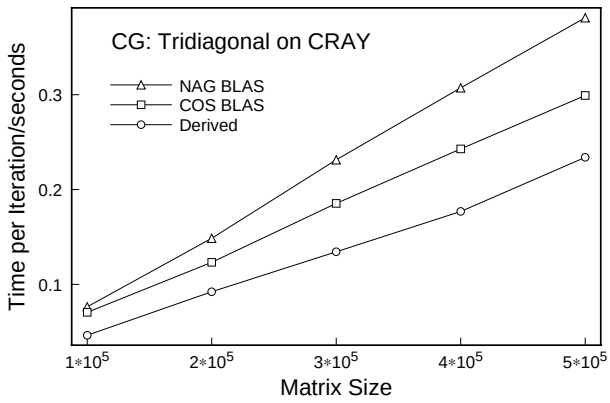

Figure 22: Execution times for one iteration of derived and BLAS implementations of CG for tridiagonal matrix on CRAY

then the price of inflexibility must be paid: 'black box' subroutines drawn from a pre-compiled library cannot readily be tailored to the particular requirements of a given program. Thus, the significance of the execution results presented here is not so much that a particular derived implementation is faster than a particular 'competing' implementation, but that the use of the derivational approach provides extra flexibility which may result in superior execution performance even though initial algorithm specifications are highly abstract.

Any such advantage in terms of execution performance gained through the derivational approach is additional to other advantages discussed previously, particularly the elegance of the implementation procedure itself and the reuse of programmer effort through reuse of sub-derivations and transformations.

For the most commonly used cases — tridiagonal and quindiagonal matrices — the BLAS implementation was slower than the derived implementation by an average of 38% and 46%, respectively. Other band parameters (various combinations having a total band width up to 32) were tested over a range of matrix sizes. In addition, a sample of band widths up to 201 were tested with a fixed matrix size. The BLAS implementation was slower by an average of from 17% to 50%. For given numbers of lower and upper diagonals, the difference in execution times for the derived and BLAS implementations varied significantly as the matrix size was varied; for example, the BLAS implementation was from 12% to 38% slower than the derived implementation.

The NAG BLAS implementation fared worse than the COS BLAS implementation: for tridiagonal and quindiagonal matrices, the NAG implementation was an average of, respectively, 67% and 95% slower than the derived implementation (with loop unrolling).

8.2.3 Other Results

Other implementations have been derived, including a DAP implementation of CG for banded matrices, a CRAY implementation of CG for row-wise matrices (the salient sections

```
val atq:real vector = transpose(a)*q;
→
      do 230 i = 1,n,1
      atq(j) = 0.0
230 continue

      do 250 i = 1,n,1
      do 240 k = 1,w,1
      j = As(i,k)
      atq(j) = atq(j)+Ap(i,k)*q(i)
240 continue
250 continue
```

```
val q′:real vector = a*r′+q*beta
→
      do 340 i = 1,n,1
      g63 = 0.0
      do 330 k = 1,w,1
      g63 = g63+Ap(i,k)*r1(As(i, k))
330 continue
      q(i) = g63+q(i)*beta
340 continue
```

Figure 23: The row-wise sparse versions of the matrix-vector product computations of CG

of which — the computations of r' and q' — are shown in Figure 23), and sequential implementations of CG for both banded and row-wise matrices (the sequential implementations are similar to the CRAY implementations).

The derived forms appeared, upon examination, to be efficient — no significant inefficiency was apparent. However, no independent implementations were available to provide a more objective assessment.

It should also be recalled that dense implementations of the algorithms have been derived for a number of architectures. The sources for the dense and sparse implementations differ only in that the latter contains an assertion of sparsity.[10]

9 Related Work

The underlying themes of the work reported here have been widely discussed in the literature: for example, functional programming [13, 14, 15] and formal program manipulation, transformation and derivation [16, 17, 18, 19, 20, 21].

Wainright and Sexton [22] discuss various data structures that can be used to represent sparse matrices in functional languages; they use a functional language as an implementation language in which a programmer expresses a program explicitly designed for sparse matrices. In contrast, the work reported here uses a functional language as a language for expressing specifications which are independent of implementation details such as sparsity.

Standish *et al* [23] gave an example of optimizing an imperative implementation of matrix multiplication for upper triangular matrices using an interactive transformation system — the programmer interactively instructs the system as to which transformations should be applied to what parts of a program; though the system then attends to the details of applying the transformations.

Bik and Wijshoff [24, 25] are developing a Fortran compiler that automatically optimizes operations on sparse matrices. Their work has several similarities to that reported here:

- Their source is a standard, dense implementation of an algorithm.
- Their optimization begins with the introduction of guarded commands (similar to the introduction of conditional expressions used here). They then manipulate programs, attempting to achieve the following:
  - to determine when commands can be discarded (for example, a command that adds zero to a variable can be discarded); this is similar to the simplification of expressions performed here;
  - to attempt to reduce the number of iterations performed by loops (for example, by restricting iterations only to those that involve non-null computations); this is similar to the restrictions of reductions performed here.

The fundamental differences between Bik's and Wijshoff's work and that reported here are:

- Their work assumes an imperative context (converting a Fortran program into object code) whereas the work reported here assumes a pure, functional context; the simple semantics of a pure, functional context makes transformation easier. However, their work is more ambitious, in some areas, than that reported here — they consider algorithms which *construct* sparse matrices and which manipulate matrices which have possibly dynamic structures (whilst still performing optimizations for matrices having static structure). They do not, however, automatically optimize programs to take advantage of symmetry.
- The program manipulation that they perform includes selecting, for the compact storage of sparse matrices, data structures appropriate for the operations to be performed on the matrices and to the architecture for which the program is being compiled. Using the method reported here, a programmer is responsible for choosing a compact data structure which is generally architecture neutral; operations on the compact store are subsequently tailored to particular architectures.
- Their work targets vector processors and sequential processors; extending it to array processors may prove difficult.

[10] The specification containing the sparse assertion may be used to derive dense implementations since the dense derivations will ignore the assertion.

The sparse compiler is still under development, so a direct comparison of the execution performance of the object code produced by the sparse compiler with that of the object code produced by a standard compiler from derived implementations is not possible.

10 Conclusions

A systematic method has been presented for tailoring high-level, abstract functional specifications of algorithms (of a certain class) to use programming techniques optimized for sparse matrices. This method has been encoded as sequences of program transformations which are applied entirely automatically. Variations of the transformations tailor specifications for different types of sparsity.

The transformations are independent of any particular computational architecture. They can easily be combined with other sequences of transformations that are designed to tailor specifications for execution on particular architectures, and with transformations designed to perform other optimizations. Such combination has been illustrated by the derivation of implementations for tridiagonal, symmetric tridiagonal, band and row-wise sparse matrices for sequential, vector and array architectures.

The effectiveness of the derivational approach has been assessed by comparing the execution efficiency of derived implementations of two non-trivial algorithms against manually constructed implementations (construction was independently performed for at least the computationally significant sections of the implementations). The derived implementations have been shown to be efficient, performing as well as the manual implementation in one case and significantly outperforming a manual implementation in a second case.

The sparse sub-derivations are currently restricted to algorithms in which sparse matrices are not constructed during the execution of the algorithm, but which are given as input data. It is known that the transformation techniques can be extended to algorithms in which elementary construction techniques are employed; for example, if an algorithm requires an input matrix to be normalized (by dividing each element by some value) and the input matrix is specified by the programmer as being sparse, then transformations can automatically determine that the normalized matrix is also sparse, and can optimize its storage and manipulation. The range of construction techniques for which transformations can automatically determine that a constructed matrix is sparse needs to be investigated.

Currently, it is assumed that there is only a single form of sparsity in any implementation. Technical details of the method may need to be altered if multiple forms of sparsity are combined.

In summary, this paper extends previous derivations in an important new direction. It adds to existing evidence that programmers can employ high-level, abstract specifications of algorithms without sacrificing execution efficiency, even when technically complex implementation methods are required to obtain the optimum performance from an implementation.

11 Acknowledgments

The work reported in this paper was supported by SERC grant GR/G 57970 and by a research studentship from the Department of Education for Northern Ireland. The authors are indebted to James Boyle for his assistance in this work.

References

[1] Boyle, J. M. and Harmer, T. J. (1992) A practical functional program for the CRAY X-MP. *Journal of Functional Programming*, **2**(1), 81–126.

[2] Boyle, J. M. (1989) Abstract programming and program transformation. In *Software Reusability Volume 1*, ACM Press/Addison-Wesley, 361–413.

[3] Boyle, J. M., Clint, M., Fitzpatrick, S. and Harmer, T. J. (1992) The construction of numerical mathematical software for the AMT DAP by program transformation. In *Proceedings of CONPAR VAPP V* (Lecture Notes in Computer Science 634), Springer-Verlag, 761–767.

[4] Wikström, A. (1987) *Functional Programming using Standard ML*. Prentice Hall.

[5] Boyle, J. M. (1970) *A transformational component for programming languages grammar*. Technical Report ANL-7690, Argonne National Laboratory, July.

[6] Clint M., Fitzpatrick, S., Harmer, T. J., Kilpatrick, P. L. and Boyle, J. M. (1994) A family of data-parallel derivations. In *Proceedings of High Performance Computing and Networking, Volume II*, (Lecture Notes in Computer Science 797), Springer-Verlag, 457–462.

[7] Fitzpatrick, S. (1994) *The Specification of Array-Based Algorithms and the Automated Derivation of Parallel Implementations through Program Transformation*. PhD thesis, Faculty of Science, The Queen's University of Belfast, http://www.cs.qub.ac.uk/~S.Fitzpatrick/Papers/PhD/.

[8] Pissanetsky, S. (1984) *Sparse Matrix Technology*. Academic Press.

[9] Key, J. E. (1973) Computer program for solution of large sparse unsymmetric systems. *Int. J. Numer. Meth Eng.*, **6**, 497–509.

[10] Weston, J. S. and Clint, M. (1990) Two algorithms for the parallel computation of eigenvalues and eigenvectors of large symmetric matrices using the ICL DAP. *Parallel Computing*, **13**, 281–288.

[11] Modi, J. J. (1988) *Parallel Algorithms and Matrix Computations*. Oxford University Press.

[12] Dongarra, J. J., DuCroz, J., Hammarling, S. and Hanson, R. (1988) An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, **14**(1), 1–17.

[13] MacLennan, B. J. (1989) *Functional Programming: Practice and Theory*. Addison-Wesley.

[14] Field, A. J. and Harrison, P. G. (1988) *Functional Programming*. (International Computer Science Series), Addison Wesley.

[15] Hughes, J. (1989) Why functional programming matters. *The Computer Journal*, **32**(2).

[16] Partsch, H. A. (1990) *Specification and Transformation of Programs*. (Texts and Monographs in Computer Science), Springer-Verlag.

[17] Burstall, R. M. and Darlington, J. (1977) A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, **24**(1), 44–67.

[18] Darlington, J., Harrison, P., Khoshnevisan, H. *et al.* (1989) A functional programming environment supporting execution, partial execution and transformation. In *PARLE '89 Parallel Architectures and Languages Europe, I*, (Lecture Notes in Computer Science 365), Springer-Verlag, 286–305.

[19] Bird, R. S. (1989) Algebraic identities for program calculation. *The Computer Journal*, **32**(2),122–126.

[20] Backhouse, R. (1989) An exploration of the Bird-Merteens formalism. In *STOP Summer School on Constructive Algorithmics*, 1–61.

[21] Back, R. J. R. and Sere, K. (1990) Stepwise refinement of parallel algorithms. *Science of Computer Programming*, **13**, 133–180.

[22] Wainright, R. L. and Sexton, M. E. (1992) A study of sparse matrix representations for solving linear systems in a functional language. *Journal of Functional Programming*, **2(1)**, 61–72.

[23] Standish, T. A., Kibler, D. F. and Neighbors, J. M. (1976) Improving and refining programs by program manipulation. In *Proceedings of the Annual Conference of the Association for Computing Machinery, '76*, ACM, 509–516.

[24] Bik, A. J. C. and Wijshoff, H. A. G. (1993) Advanced compiler opimizations for sparse computations. In *Proceedings of Supercomputing 93*, 430–439.

[25] Bik, A. J. C. and Wijshoff, H. A. G. (1993) Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, 416–424.

A Derived Implementations of POT and Conjugate Gradient

Figures 24 and 25 show implementations of Conjugate Gradient and POT. The following points should be noted:

- The comments have been manually inserted and variables manually renamed to enhance readability.
- In the POT implementation (for the DAP), the '*'s in array declarations indicate that the elements of the arrays are to be processed in parallel.
- In the POT implementation, the symmetric tridiagonal matrix A whose eigensystem is to be computed is realized as a pair of vectors Adiag and Asub, rather than as a matrix with 2 columns. The latter would not permit efficient use of the DAP processor grid. (This realization was introduced by a sub-derivation that has not been discussed in this paper.)

```
      real U(*n,*n), Asub(*n), Adiag(*n), B(*n,*n)
      real Bdiag(*n), AU(*n,*n)
      real col(*n), GS(*n,*n), project(*n,*n)
      real bb, g1164(*n)
      logical mask(*n,*n), colmask(*n,*n)
      integer order(*n)
100   continue

C     Compute B
      AU = matc(Asub,n)*shsp(U)
     .   +matc(Adiag,n)*U
     .   +matc(shlp(Asub,1),n)*shnp(U)
      B = 0.0
      do 101 i = 1,n,1
      B = B+matc(U(i,),n)*matr(AU(i,),n)
101   continue

C     Test for convergence
      Bdiag = B(diagpat,)
      g1164 = abs(Bdiag)
      bb = (sum(abs(B))-sum(g1164))/(n*(n-1))
      if (bb.lt.1.0E-15) then
      goto 130
      endif

C     Compute transform(B)
      call indexvec(g1227)
      GS = 0.0
      D = matr(Bdiag,n)-matc(Bdiag,n)
      denom = D
     .   +merge(-1.0,1.0,D.lt.0.0)*sqrt(D*D+4*B*B)
      T = merge(1.0,
     .   2*B/denom,
     .   abs(denom).lt.1.0E-14)
      T(diagpat) = 1.0

C     Compute A*U*transform(B)
      do 102 i = 1,n
      GS = GS+matc(AU(,i),n)*matr(T(i,),n)
102   continue

C     Sort eigenvalues
      colmask = .true.
      g1240 = matr(g1164,n)
      g1141 = matc(g1164,n)
      g1240 = merge(1,
     .   merge(1,0,(g1240.eq.g1141)
     .     .and.patlowertri(n)
     .     .and..not.diagpat),
     .   g1240.gt.g1141)
      order = coln(matc(g1227,n)
     .   .eq.matr(sumc(g1240)+1,n))
      j = 1

C     Orthogonalize
110   continue
      if (j.gt.n) then
      goto 120
      else
      col = GS( ,order(j))
      col = col/sqrt(sum(col*col))
      if (j.eq.n) then
      GS(,order(j)) = col
      j = j+1
      else
      mask = colmask
      mask(order(j)) = .false.
      project = matc(col,n)
      project = matr(sumr(GS*project),n)*project
      GS(mask) = GS-project
      GS( ,order(j)) = col
      j = j+1
      colmask = mask
      endif
      goto 110
      endif
120   continue
      U = GS
      goto 100

130   continue
```

Figure 24: DAP implementation of POT for matrix A symmetric tridiagonal

```
      real Ap(n,l+u+1)
      real x(n), r(n), p(n), q(n)
      real alpha, beta

100 continue
C     Test for convergence
      rr = 0.0
      do 110 i = 1,n,1
      rr = rr+r(i)*r(i)
110 continue
      if (sqrt(rr).lt.1.0E-14) then
      goto 500
      endif

C     Compute alpha
      alpha = 0.0
      do 120 k = 1,n,1
      alpha = alpha+q(k)*q(k)
120 continue
      alpha = rr/alpha

C     Compute r'
      do 140 i = n-l+1,n,1
      t = 0.0
      do 130 k = i-u,n,1
      t = t+Ap(k,i+l+1-k)*q(k)
130 continue
      r1(i)= r(i)-t*alpha
140 continue

      do 160 j = 1,u,1
      t = 0.0
      do 150 k = 1,j+l,1
      t = t+Ap(k,j+l+1-k)*q(k)
150 continue
      r1(j)= r(j)-t*alpha
160 continue

      do 180 i = u+1,n-l,1
      t = 0.0
      do 170 k = i-u,i+l,1
      t = t+Ap(k,i+l+1-k)*q(k)
170 continue
      r1(i)= r(i)-t*alpha
180 continue

C     Compute beta
      t = 0.0
      do 190 i = 1,n,1
      t = t+r1(i)*r1(i)
190 continue
      beta = t/rr

C     Compute x'
      do 200 i = 1,n,1
      x(i)= x(i)+p(i)*alpha
200 continue

C     Compute p'
      do 210 i = 1,n,1
      p(i)= r1(i)+p(i)*beta
210 continue

      do 220 i=1,n,1
      r(i) = r1(i)
220 continue

C     Compute q'
      do 240 i = n-u+1,n,1
      g97 = 0.0
      do 230 k = i-l,n,1
      g97 = g97+Ap(i,k+l+1-i)*r1(k)
230 continue
      q(i)= g97+q(i)*beta
240 continue

      do 260 i = 1,l,1
      g97 = 0.0
      do 250 k = 1,i+u,1
      g97 = g97+Ap(i,k+l+1-i)*r1(k)
250 continue
      q(i)= g97+q(i)*beta
260 continue

      do 280 i = l+1,n-u,1
      g97 = 0.0
      do 270 k = i-l,i+u,1
      g97 = g97+Ap(i,k+l+1-i)*r1(k)
270 continue
      q(i)= g97+q(i)*beta
280 continue

      goto 100

500 continue
```

Figure 25: CRAY implementation of CG for band matrices