A Dimension Abstraction Approach to Vectorization in Matlab

Neil Birkbeck Jonathan Lévesque José Nelson Amaral
University of Alberta
Edmonton Alberta, Canada T6G 2E1
{birkbeck,jrl2,amaral}@cs.ualberta.ca

Abstract

Matlab is a matrix-processing language that offers very efficient built-in operations for data organized in arrays. However Matlab operation is slow when the program accesses data through interpreted loops. Often during the development of a Matlab application writing loop-based code is more intuitive than crafting the data organization into arrays. Furthermore, many Matlab users do not command the linear algebra expertise necessary to write efficient code. Thus loop-based Matlab coding is a fairly common practice. This paper presents a tool that automatically converts loop-based Matlab code into equivalent array-based form and built-in Matlab constructs. Array-based code is produced by checking the input and output dimensions of equations within loops, and by transposing terms when necessary to generate correct code. This paper also describes an extensible loop pattern database that allows user-defined patterns to be discovered and replaced by more efficient Matlab routines that perform the same computation. The safe conversion of loop-based into more efficient array-based code is made possible by the introduction of a new abstract representation for dimensions.

1. Introduction

Matlab™ is an algebraic solution environment optimized for matrix operations that is broadly used for scientific and engineering computations. Because Matlab is an interpreted high-level language, computations expressed as nested loops are quite slow. On the other hand, it is often simpler to write programs that operate on single pieces of data rather than on matrices. Thus, Matlab programs are often written as nested loops by programmers that either lack the training on array-based programming or are unaware of the performance differences involved. An automatic vectorization tool could deliver the performance of vectorized code to such programmers.

Matlab is a widely-used high-level mathematical modelling tool that is often employed by engineers that are experts on the application field rather than on computer programming. The premise of this work is that, whenever possible, automatic tools should detect computations for which intrinsic Matlab functions are available and replace them by their more efficient counterparts. This paper first describes the conversion of loop-based code to array-based code, then it introduces an extensible loop pattern database that identifies pre-defined patterns in the program and replaces them by more efficient Matlab intrinsic functions. An extensive loop pattern database has the potential of significantly increasing engineering productivity without requiring the retraining of Matlab programmers. Because the less efficient code written by a novice programmer is likely to be less terse and more easily understood by other programmers, such tools would also facilitate the maintenance of the code. Although we focus our attention on vectorization of Matlab loops, our framework extends to any language having efficient intrinsics that are equivalent to less efficient code. The framework is useful wherever parts of an abstract syntax tree may be replaced with more efficiently implemented versions. For instance, contemporary processors could benefit from the replacement of code portions by SIMD instructions.

The software tool presented in this paper performs source-to-source transformations of Matlab programs. The loop vectorizer examines each loop in the original code to determine if it can be vectorized. Some loops cannot be vectorized due to loop-carried dependencies. The loops that can be vectorized are replaced by their equivalent array-based form, which speeds up execution most of the time. The contribution of this paper is a new method to automatically generate code that is optimized for Matlab from general code. The method is based upon low-level components, such as dimensions and operators, extracted from a parsing tree representation of the program.

The main contributions of this paper are:

- An extension of Allen & Kennedy's [1] *codegen* algorithm, giving it the power to correctly vectorize accumulator variables as well as cases where a row vector

is added to a column vector element-wise.

- A new abstract representation for the dimensions of Matlab variables that leads to an elegant framework for the automatic vectorization of loop-based Matlab code.
- A new extensible loop pattern database that allows the replacement of general pre-defined patterns by Matlab intrinsic functions.

Section 2 presents the abstract representation of dimensions and the framework for detecting vectorization opportunities. Section 3 describes the framework for the extensible loop pattern database. The integration of both techniques in Allen & Kennedy's codegen algorithm is described in Section 3.2. Section 4 discusses the implementation of a prototype that demonstrate the viability of the ideas presented in this paper. Section 6 discusses related work.

2. Dimensionality Abstraction

This paper introduces a new automatic vectorization tool that is based on the *codegen* algorithm presented by Allen & Kennedy [1]. Their algorithm partitions a data dependence graph (DDG) into strongly connected components (SCC). The edges of the original DDG connect the SCCs. The algorithm then visits these SCCs in topological order according to these edges. An SCC can be a single node with no recurrences on itself. Single-node SCCs can be vectorized immediately. For multi-node SCCs or for single-node SCCs that have a recurrence, the outermost loop is run sequentially, and then the edges within the SCC that are carried by the outermost loop are removed. The code generation procedure is then called on this simpler graph, but now the vectorization can only occur on the remaining inner loop nest.

When vectorizing a statement, the original *codegen* algorithm replaces the loop index variable by the corresponding loop range (e.g., occurrences of `i` are replaced with `1:n` for a loop that runs from `i=1` to `i=n`). While this simple replacement is correct for pointwise *vector* operations, it may introduce errors in the vectorization of Matlab loops. In Matlab a vector has an associated type that specifies its orientation: row vector or column vector. Thus the vectorizer must model the dimensions of all the variables involved in all the statements in the loop to generate correct code.

Dimension information is also required to determine the semantics of a particular statement. For instance, consider the statement `x(i)=y(i,h)*z(h,i)` in a loop with index variable `i` (see Appendix A for a concise overview of the Matlab notations used in this paper). This statement has two different semantic meanings depending on whether `h` is a scalar or a vector. If `h` is a scalar then the statement is performing a product of scalars, and upon vectorization should remain as a pointwise operation — `x(1:n)=y(1:n,h).*(z(h,1:n)')`. On the other hand, if `h` is a vector then the statement is performing a dot product between a row vector and a column vector, which means that each element of `x(i)` will be the result of a dot product of vectors instead of a product of scalars and should be vectorized as matrix multiplication — `x(1:n)=y(1:n,h)*z(h,1:n)`.

Having motivated the need for dimensionality analysis, the remainder of this section describes this analysis and then describes how it can also be used to vectorize non-pointwise operations.

2.1. Abstraction of Dimensions

The goal of the dimensionality analysis is to discover whether dimensions of expressions will be legal if vectorization occurs. For this analysis we introduce an abstraction of the dimensions of variables, where the size of a variable in a given dimension is represented by one of two symbols: a 1 representing that the size of the variable is 1 in that dimension; or *, meaning that the size of the variable is greater than 1 in the given dimension. The dimensionality of a variable v is represented in an ordered list of symbols, $dim(v) = (a_1, a_2, \ldots, a_n)$ where $a_n \in \{1, *\}$. The i^{th} element corresponds to the abstract size of the variable in the i^{th} dimension. The following examples illustrate the use of abstract representation for dimensions:

- the dimensions of a scalar (e.g., a 1×1 matrix) is represented as $(1, 1)$ or simply (1)
- the dimensions of a $1 \times n$ row vector is represented as $(1, *)$
- the dimensions of a $m \times 1$ column vector is represented as $(*, 1)$
- the dimension of a $k \times l$ matrix is represented as $(*, *)$

Our analysis requires this knowledge of shape for any variables in the code being vectorized. The dimensionality analysis is intuitive: if the dimensions of operands in expressions agree after a simple index-variable replacement, then the vectorization is correct and should be allowed. To give a precise definition, the simple abstract representation of dimensionality must be extended. First, we introduce the concept of *vectorized dimensionality*, which is an abstract representation of the dimension of an expression *after* vectorization of one or more loop index variables has occurred. The vectorized dimensionality of an expression e with respect to index variable i is denoted $dim_i(e)$. The elements of the vector $dim_i(e)$ may be one of $\{1, *, r_i\}$, where r_i is a special symbol relating the size of a dimension to the loop

range for index variable i. Like the "$*$" symbol, an r_i symbol also represents some number greater than 1. The rules for computing the vectorized dimensionality of a simple index expression are extracted from the Matlab programming language itself (see Table 1 for an outline of these rules). The notation $dim_{i,j}(e)$ refers to the vectorized dimensionality of expression e after replacing the index variables by the vector ranges (*i.e.*, $dim_{i,j}(e) = dim_i(dim_j(e))$). Furthermore, unless specified otherwise, dim_i is a shorthand notation denoting the vectorized dimensionality after expanding all loop index variables.

For example, if a loop iterates on $i = 1 : n$, then after the expansion of the index variable i, the expression i will be a row vector of length $1 \times n$, hence $dim_i(i) = (1, r_i)$. Similarity, if $dim(A) = (*, 1)$ then $dim(A(i)) = (1, 1)$, but the dimensionality after replacing i with the loop range will be $dim_i(A(i)) = (r_i, 1)$ because $A(1 : n)$ is a column vector with size $n \times 1$.

The concept of vectorized dimensionality is similar to that of a *type* in type inference. In contrast, the vectorized dimensionality is an abstract representation of the shape of an expression if a loop containing the expression was vectorized. Similar to type inference, the dimensionality analysis consists of propagating these vectorized dimensionalities up the parse tree, but the goal is different from type inference: this abstraction is used to determine when an optimization can be performed as opposed to attempting to determine the *type* of a variable.

With this formal definition of the vectorized dimensionality of simple expressions (e.g., constants, identifiers, and subscripted expressions), we can define the compatibility rules for expressions that may prevent vectorization, such expressions are pointwise arithmetic expressions and assignment expressions. Two vectorized dimensionalities $dim_i(e_1)$ and $dim_i(e_2)$ are equal, $dim_i(e_1) \equiv dim_i(e_2)$, if and only if all their elements are identical. Two vectorized dimensionalities are compatible, $dim_i(e_1) \simeq dim_i(e_2)$, when their reduced vectorized dimensionalities are equal, $f_{reduce}(dim_i(e_1)) \equiv f_{reduce}(dim_i(e_2))$. The reduction function f_{reduce} removes any trailing 1 dimensions from its vectorized dimensionality input (for example, a 5x5 matrix is effectively the same as a 5x5x1 matrix). It should be noted that although r_i is similar to `*`, the two symbols are not compatible. Furthermore, r_i is always incompatible with r_j for $i \neq j$, even in the case that i and j have the same loop bounds.

An assignment expression of the form $e_{lhs} = e_{rhs}$ is compatible when either of the following hold:

1. $dim_i(e_{lhs}) \simeq dim_i(e_{rhs})$;
2. e_{rhs} is a scalar.

For the pointwise arithmetic operators, $e = e_{lhs} \odot e_{rhs}$, $\odot \in \{+, -, .*, ./, .\hat{}\}$, we must also define $dim_i(e)$. The pointwise operator is compatible when one of the conditions below holds. The dimension of the expression in the left-hand side of the statement is set as indicated.

1. $dim_i(e_{lhs}) \simeq dim_i(e_{rhs})$. Set $dim_i(e_{lhs} \odot e_{rhs}) = dim_i(e_{lhs})$;
2. e_{lhs} is a scalar. Set $dim_i(e_{lhs} \odot e_{rhs}) = dim_i(e_{rhs})$;
3. e_{rhs} is a scalar. Set $dim_i(e_{lhs} \odot e_{rhs}) = dim_i(e_{lhs})$.

These rules allow for a scalar to be assigned to a matrix/vector and also allow for the scalar to be an operand of a pointwise binary operator where the other operand is a matrix/vector, which is perfectly legal in Matlab. If the above rules do not hold for any pointwise operator or assignment in a statement, then the statement must not be vectorized. The use of the notion of compatibility protects the code semantics by disallowing transformations whose bounds match but which are not equivalent to the original source code.

2.2. Recognizing Transposes

The compatibility rules described in Section 2.1 are quite restrictive and will not allow for vectorization of the simple statement `z(i)=x(i)+y(i)` when `x` is a column vector and `y` is a row vector. A straightforward extension to the compatibility rules identifies when a transpose can make the operands of an expression compatible. If an assignment is found to be incompatible and $dim_i(e_{lhs}) \simeq f_{reverse}(dim_i(e_{rhs}))$, then the assignment of $e_{lhs} = e'_{rhs}$ is compatible and vectorization is permitted. The rules are similar for pointwise expressions: $dim_i(e_{lhs}) \simeq f_{reverse}(dim_i(e_{rhs}))$ or $f_{reverse}(dim_i(e_{lhs})) \simeq dim_i(e_{rhs})$ will produce a compatible expression.

An example that illustrates the rules for assignment and a pointwise operator is given below.

```
for i=1:m
  for j=1:n
    A(i,j)=B(j,i)+C(i,j);
  end
end
```

In this example, $dim_{i,j}(\texttt{B(j,i)}) = (r_j, r_i)$ and $dim_{i,j}(\texttt{C(i,j)}) = (r_i, r_j)$ are not compatible. The extended analysis verifies that if `C(i,j)` is transposed, then the statement `B(j,i)+C(i,j)'` is compatible. This transposition results in $dim_{i,j}(\texttt{B(j,i)} + \texttt{C(i,j)}') = dim_{i,j}(\texttt{B(j,i)}) = (r_j, r_i)$. After the transposition $dim_{i,j}(\texttt{A(i,j)}) = (r_i, r_j)$, implying that the entire right hand side of the assignment must also be transposed, resulting in the following vector statement:

Expression Type	$dim_i(e)$
Scalar	(1)
Identifier	$(1, r_i)$ if identifier name is "i" $dim(e)$ otherwise
Colon Expression (e.g., $1:3:n$)	$(1, *)$
Subscripted Expression (e.g., $M(e_1, e_2, \ldots, e_k)$)	$dim_i(e_1)$ if k==1 and (isMatrix(M) or isMatrix(e_1)) $(f_{max}(dim_i(e_1)), \ldots, f_{max}(dim_i(e_k)))$ otherwise
Signed Expression (e.g.,$+\hat{e}$,$-\hat{e}$)	$dim_i(\hat{e})$
Transposed Expression (e.g.,$\hat{e}'$)	$f_{reverse}(dim_i(\hat{e}))$

Table 1. The rules for computing vectorized dimensions of simple expressions. The predicate isMatrix(M) is true if and only if M is a multi-dimensional matrix. The function f_{max} returns the largest dimension of a vector argument (e.g., $f_{max}(1, *) = f_{max}(*, 1) = *$, $f_{max}(1, 1) = 1$, $f_{max}(1, r_i) = f_{max}(r_i, 1) = r_i$). The function $f_{reverse}$ returns the dimension vector that is the reverse of its argument.

```
A(1:m,1:n)=(B(1:n,1:m)+C(1:m,1:n)')'
```

A later optimization, not investigated in this paper, would identify that the transpose can be distributed to generate a simpler equivalent form:

```
A(1:m,1:n)=B(1:n,1:m)'+C(1:m,1:n)
```

This example also demonstrates why the special index variable symbols r_i and r_j must be treated as different. Consider the case where `m = n`, *i.e.* the loops iterate through the same range. If the analysis were to treat r_i and r_j as the same symbol for the computation of the equivalence of vectorized dimensionality it would obtain $dim_{i,j}(\texttt{B(j,i)}) \equiv dim_{i,j}(\texttt{C(i,j)})$, which would result in the transpose *not* being inserted and would produce an incorrect vectorization.

3. An Extensible Loop Pattern Database

Certain built-in Matlab functions can provide extremely efficient computation involving multiple data, but the vectorization algorithm cannot make use of them without knowledge of their functionality and use. The techniques described in Section 2 deal exclusively with the correct vectorization of pointwise operations, but will fail to vectorize any statement that has an incompatible vectorized dimensionality (e.g., loop-based matrix multiplication). To improve vectorization in such cases we encode knowledge of possible transforms in a pattern database which is capable of resolving the obstructing dimensionality disagreements. These transformations can make use of native intrinsic functions, such as the `repmat` function, to enhance vectorization potential. This extensible database framework permits plugin-style addition of routines vectorizing an operator matching a specified definition.

For binary arithmetic operations, a number of transformations are stored in an extensible database. Each transformation is indexed by a unique pattern identified by the dimensionality of its operands — $dim_i(e_{lhs})$, $dim_i(e_{rhs})$ — and by the type of operator. Each transformation also has a corresponding output dimension, $dim_i(e_{out})$. When the compatibility checking discussed in Section 2.1 fails, the vectorizer checks to see if any transformation in the database has a pattern that matches both the current operator and the current dimensionality of the left and right-hand side operands. If a match is found, the expression is associated with the transformation, and $dim_i(e_{out})$ of the matched transformation is returned as the resulting vectorized dimensionality of the expression. If the statement containing this expression is found to be compatible, then upon vectorization any transformations associated with the expression in the statement are applied.

This database is extensible. Users may add their own patterns and methods for handling them as necessity demands. A sample database with three pattern-based transformations, already implemented, is displayed in Table 2. The user defines a new vectorization opportunity by specifying the operator, the dimensions of all terms in the equation, and the desired replacement code.

The first pattern in Table 2 is a matrix product. The pattern matches the loop code because $dim_i(\texttt{X(i,:)}) = (r_i, *)$ and $dim_i(\texttt{Y(:,i)}) = (*, r_i)$ and the operator is *. When checking that the assignment has compatible dimensions, the pattern is recognized and the right-hand side of the assignment expression has a dimensionality of $dim_i(e_{out}) = (1, r_i)$, which agrees with $dim_i(\texttt{a(i)}) = (1, r_i)$ and makes the assignment compatible. When vectorization occurs, the transformation is applied to `X(i,:)*Y(:,i)` giving `sum(X(1:n,:)'.*Y(:,1:n))`. The transformed code takes the pointwise product of the transpose of the first matrix and of the second matrix and then computes the sum along the rows. This computation places the dot product of the `i`-th row of `X` and `i`-th column of `Y` in the `i`-th column

Pattern ID	Pattern				Loop Code	Vector Code
	$dim_i(e_{out})$	$dim_i(e_{lhs})$	Operator	$dim_i(e_{rhs})$		

1	$(1, r_1)$	$(r_1, *)$	*	$(*, r_1)$	`for i=1:n,`
					`  a(i)=X(i,:)*Y(:,i);`
					`end`
						`a(1:n)=sum(X(1:n,:)'`
						`  .*Y(:,1:n))`

2	(r_1, r_2)	(r_1, r_2)	$\odot$	$(r_1, 1)$	`for i=1:m`
					` for j=1:n`
					`  A(i,j)=B(i,j)+C(i);`
					` end`
					`end`
						`A(1:m,1:n)=B(1:m,1:n)+`
						`  repmat(C(1:m),`
						`  1,size(1:n,2))`

3	$(1, r_1)$	(r_1, r_1)	$(\cdot)$	–	`for i=1:n`
					` a(i)=A(i,i)*b(i);`
					`end`
						`a(1:n)=A((1:n)+`
						`  size(A,1)*`
						`  ((1:n)-1))`
						`  .*b(1:n)`

Table 2. Examples of Patterns in the Pattern Database.

of the output as was done by the original statement.

Patterns may contain more than one loop index variable. An example of such a pattern on any pointwise operator is the second pattern in Table 2. For this pattern, the analysis presented in Section 2.1 would vectorize the `j` loop but not the `i` loop. On the other hand, when the transformation corresponding to this pattern is applied it produces the single vector statement shown in the vector-code column. This single statement duplicates the column vector `C` `n`-times using `repmat`, producing a matrix with the same dimensions of `B`.

These pattern-based transformations are also useful in the vectorization of patterns resulting from subscripted matrix expressions. In particular, if patterns can be matched after the creation of the vectorized dimension for a matrix subscript then the analysis can detect patterns such as accesses to the diagonal of a matrix. Consider the simple loop in the third pattern in Table 2. This loop is vectorizable if the access to `A(i,i)` can be reduced to a vector statement. The vectorized dimensionality of `A(i,i)` is (r_1, r_1) but `A(i,i)` is accessing a one-dimensional region of `A`. To handle these cases, the pattern database allows for a class of transformations that match patterns on matrix accesses (e.g., in the database the operator is $(\cdot)$). This class of transformations is used to transform a matrix access with vectorized dimensionality $dim_i(e)$ into an equivalent mathematical expression with a vectorized dimensionality $dim_i(e_{out})$ containing no duplicate $r_{i_1}, r_{i_2}, \cdots, r_{i_k}$ symbols. The diagonal example transforms the vectorized dimensionality of (r_1, r_1) to $(1, r_1)$. The transformation that needs to be applied simply replaces the double subscript by a single subscript and accesses the matrix as a vector using the fact that matrices in Matlab are stored in column-major format. For example, assume that the matrix is accessed using a single index variable in both dimensions, which may take the form `A(`c_1`*i+`c_2`,`c_3`*i+`c_4`)`, where c_i are scalars. The corresponding vector access is `A(`c_1`*i+`c_2`+size(A,1)*(`c_3`*i+`c_4`-1))`. This is the transformation that is applied to the parse tree when the (r_1, r_1) matrix-access pattern is matched and the corresponding statement is vectorized.

3.1. Additive Reductions

The techniques discussed this far are suited to *natural statements* where the vectorized dimensionality of the statement contains an entry for each index variable. Although the class of such natural statements are useful for a Matlab vectorizer, there exist other vectorizable statements that cannot be coerced into the form of a natural statement. One important set of statements are *additive-reduction* statements, which are statements that use a loop variable to perform an accumulation. An additive-reduction statement within a loop nest with index variables $\mathtt{I} = \{\mathtt{i_1}, \mathtt{i_2}, \ldots, \mathtt{i_k}\}$ takes the following form:

$$\mathtt{A(J)} = \mathtt{A(J)} + \mathtt{E}; \quad (1)$$

where $\mathtt{A(J)}$ is the *accumulator variable* indexed by $\mathtt{J} \subset \mathtt{I}$ and `E` is any allowable expression. The index variables in

the non-empty set $\mathtt{I} - \mathtt{J}$ are said to be the *reduction* variables because they are the index variables used to perform the accumulation.

This class of statements can be properly vectorized with a simple extension of the techniques discussed above. For each expression the vectorizer maintains and propagates the *vectorized dimensionality* through a traversal of the expression's parse tree. In addition to the vectorized dimensionality, the vectorizer also maintains the set of index variables that have already been reduced for an expression during the traversal, denoted by $\rho(E)$.

Upon vectorization of an additive-reduction statement, a reduction operator is required to perform the accumulation because the statement is no longer within the loop. For any expression e with vectorized dimensionality $dim_i(e) = \{S_1, S_2, \ldots, S_n\}$, the vectorizer can perform a reduction along index variable r_i, denoted $\Gamma(E, r_i)$. Informally, this operator is an abstract representation of the effects of accumulating expression e by iterating on index variable i. The reduction operator Γ is defined as follows. If a single S_j is r_i then the reduction replaces S_j with 1, replaces e by $e' = sum(e, j)$, and replaces $\rho(e)$ with $\rho(e) \bigcup r_i$. If there is no $S_j = r_i$ then e is replaced with $tripcount(r_i) * e$ and $\rho(e)$ becomes $\rho(e) \bigcup r_i$. Throughout the dimensionality-checking process the reduction operator is selectively applied to ensure that the dimensionality and reduced variables of expressions are consistent. For example, for any statement of the form of equation 1 the vectorizer must ensure that $dim_{i_1,i_2,\ldots,i_k}(A(J)) \simeq dim_{i_1,i_2,\ldots,i_k}(E)$ and that $\rho(E) = I - J$. If the second criterion fails then the vectorizer must use the Γ operator to reduce any variables in $I - J$ that are not in $\rho(E)$.

In fact, the dimensionality checking proceeds in much the same manner as it had in Section 3. Although with reduction statements it is possible to take advantage of the semantics of native matrix multiplication to implicitly perform reductions without the use of the Γ operator. This transformation is applicable for a binary expression $e = e_{lhs} * e_{rhs}$. If $dim_i(e_{lhs}) = (r_i, r_k)$, $dim_i(e_{rhs}) = (r_k, r_j)$, and r_k is a reducing index variable, then using the semantics of matrix multiplication the vectorizer can compute $dim_i(e) = (r_i, r_j)$ and $\rho(e) = \rho(e) \bigcup \{r_k\}$.[1]

For any other binary expression, because it is possible for the left- and right-hand sides to have non-empty reduced variable sets, the vectorizer must ensure that these variable sets agree. If the binary expression is of the form $e = e_{lhs} \pm e_{rhs}$ then the vectorizer must ensure that $\rho(e_{lhs}) = \rho(e_{rhs})$ before vectorization. This is easily accomplished by applying $\Gamma(e_{lhs}, r_{rhs})$ (resp. $\Gamma(e_{rhs}, r_{lhs})$) for r_{rhs} (resp. r_{lhs}) such that $r_{rhs} \in \rho(e_{rhs}) \wedge r_{rhs} \notin \rho(e_{lhs})$ (resp. $r_{lhs} \in \rho(e_{lhs}) \wedge r_{lhs} \notin \rho(e_{rhs})$). For any other operator the reduced variable check is successful if and only if $S \in \rho(e_{lhs}) \Rightarrow S \notin dim_i(e_{rhs})$ and $S \in \rho(e_{rhs}) \Rightarrow S \notin dim_i(e_{lhs})$. This constraint states that any index variable that is reduced in one operand cannot appear in the vectorized dimensionality of the other operand.

The associative structure of groups of multiplications in the parse tree may impede reductions through native matrix multiplication. This problem can be overcome by rearranging the structure of the parse tree into an equivalent form by utilizing the associativity of scalar and matrix multiplication[2].

3.2. Integration with the *codegen* Algorithm

Algorithm 1: CodeBlock **codegen_dim**(Graph DDG, int level, int maxlevel, LoopHeader * loopHeaders)

```
1  CodeBlock block;
2  {{Π_1, Π_2, ..., Π_n}, E_SCC}=
3        getStronglyConnectedComponents(DDG);
4  foreach Π_i (in topological order according to E_SCC) do
5   if Π_i is acyclic then
6    CodeBlock * block_ptr=&block;
7    for l=level; l < maxlevel; l++ do
8     if vectDimsOkay(Π_i,loopHeaders,l,maxlevel) then
        /* Apply t-formations                    */
9       Π_i=performTransformations(Π_i);
10      CodeBlock * vcode=
11        genVectorCode(Π_i,loopHeaders,l,maxlevel));
12      block_ptr→append(vcode);
13      break;
14    else
        /* Run loop l sequentially, add new
           code to its body                      */
15      ForLoop * lp=new ForLoop(loopHeaders[l]);
16      block_ptr→append(lp);
        /* subsequent code goes in 'lp'          */
17      block_ptr=&(lp→body);
18    end
19   end
     /* No vectorization possible:               */
20   if l==maxlevel then block_ptr→append(Π_i);
21  else
22   Graph DDG_Π=
23       removeDependenciesCarriedByLevel(Π_i,level);
24   CodeBlock * nestedBody=
25       codegen(DDG_Π,level+1,maxlevel,loopHeaders);
26   block.append(ForLoop(loopHeaders[level],nestedBody));
27  end
28  return block
29 end
```

[1] The pointwise operators are given priority over reduction by matrix multiplication.

[2] Although a more sophisticated approach may be possible, in our prototype we enumerate all possible associative groupings of the multiplications in an expression until the dimension checking is successful.

The extended dimensional analysis, including the pattern-based transformations, are easily integrated into Allen & Kennedy's *codegen* algorithm [1]. The modification to the original algorithm occurs prior to the generation of vectorized code for the acyclic blocks (see Algorithm 1). The rest of the algorithm — lines 1-5 and 21-27 — are the same as the original algorithm. The algorithm takes the following as input : *DDG*, the data dependence graph of the loop to be vectorized; *level*, an index into the outermost loop nest to consider for vectorization; *maxlevel*, the number of loops in the nest; and *loopHeaders*, an array of data structures (of length *maxlevel*) containing information about the loop index variable and the bounds of each loop in the next. Each node in the *DDG* corresponds to a statement in the original loop nest, and each node retains the parse tree for its statement.

Before generating vector code (line 7), the *vectDimsOkay* function traverses the parse tree for the single statement of the acyclic block. This function computes and propagates the vectorized dimensionalities using the rules discussed in Sections 2.1 and 2.2. During this traversal, any expressions matching the patterns in the database are marked for subsequent transformation and the output dimensionality of the pattern is propagated as the vectorized dimensionality of the matching expression. This function will fail if the statement contains a binary operator or assignment whose operands have incompatible dimensions. When this dimension checking succeeds, the marked code transformations are then applied to the statement (line 8) and the index variables are substituted to get the correct vectorized code (line 11). At this point the processing on this block is complete (line 12) and control returns to the outermost loop where the next SCC is processed.

The dimension checking in line 7 fails when the statement contains a binary operator with incompatible operands. In such cases, the outermost loop must be run sequentially (lines 14-16). The dimension checking must then be performed again, but this time only the remaining loops are considered for vectorization and only their index variables are used in the dimension checking. This process is iterated until the dimension checking succeeds. If the dimension checking is never successful, each loop in the loop nest will be run as a sequential loop.

The integration of these techniques in this manner ensures that statements will be pulled out of as many loops as possible when being vectorized, which is a property of the original algorithm. Similarly, it also ensures that statements that cannot be vectorized will remain in the loop nest.

4. A Prototype Implementation

A prototype implementation of the vectorizer and loop pattern database demonstrates that these ideas work in practice. In this prototype, the input to the vectorizer is a parse-tree representation of the Matlab source code provided by the open-source Octave [7]. Each loop nest is analyzed independently. Only for-loops are considered since they run through a specified number of iterations. Loops containing conditional statements or writing to their own index within the loop are not candidates for vectorization. Before the vectorization analysis, index variables are normalized and the data dependence analysis described by Allen & Kennedy is performed to generate a data-dependence graph. This graph is the input to the data-dependence-based vectorizer (Section 2), which produces a modified parse tree that is passed to a code generator. This sequence of processes is illustrated in Figure 1.

The dimensionality analysis requires an abstract representation of the dimension of variables in the source program. Other researchers have proposed methods for the automatic extraction of this information from the source code [5, 18]. For this prototype we assume that the output of these methods is available as annotations in comments that start with `%!`. To inform the vectorizer that `i` is a scalar, `a` is a row vector, `b` is a column vector, and that `A` is a matrix, the source code would contain the following annotation:

```
%!  i(1) a(1,*) b(*,1) A(*,*).
```

The pattern-based transformations discussed in Section 3 are implemented in a modular manner. Each pattern-based transformation is stored in its own dynamically loadable library (DLL), which defines the type of transformation, the pattern to be detected, and the definition of a function that applies the transformation as shown in the example in Figure 2. This modular design allows user-defined transformations, that extend the functionality of the vectorizer, to be dynamically loaded at runtime.

5. Experimental Results

The evaluation of this source-to-source transformation suffers from the usual lack of extensible benchmark programs for new techniques. Most users of Matlab do not publish their source code. Thus to evaluate the prototype described in Section 4, we resorted to applying it to several examples that we found around the department. The dimensional analysis approach was capable of vectorizing all the inputs for which it was applicable. We then vectorize the program examples provided by Menon & Pingali [17]. The resulting speedup from vectorization is impressive. The experimental results were all obtained by averaging 100 runs on Matlab 7.2.0.283 with the resulting source code on a 3.0 GHz Pentium D Processor. Underlying matrix operations, implemented in Matlab, may leverage common SIMD instructions but even in this case the results show speedups on the same target (Matlab).

First we demonstrate the vectorization of a typical im-

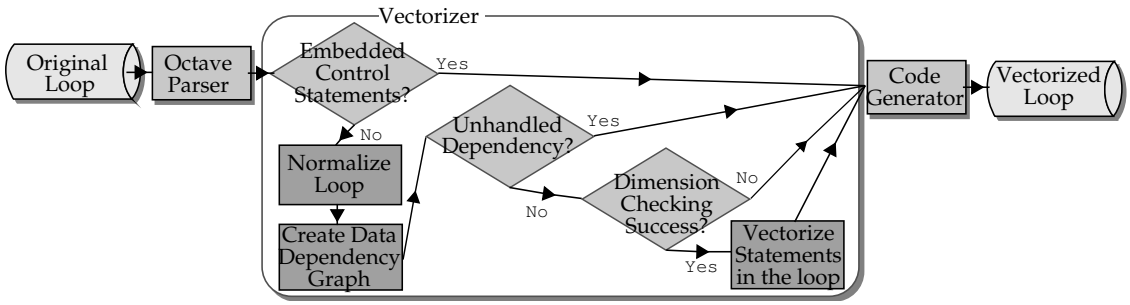

Figure 1. Information Flow Chart

```
%!im(*,*) im2(*,*) heq(1,*) h(1,*)
h=hist(im(:),[0:255]);%histogram
heq=255*cumsum(h(:))/sum(h(:));
for i=1:size(im,1),
  for j=1:size(im,2),
    im2(i,j)=heq(im(i,j)+1);
  end
end
```

```
h=hist(im(:),[(0:255)]);
heq=255*cumsum(h(:))/sum(h(:));
im2(1:size(im,1),1:size(im,2))=...
     heq(im(1:size(im,1),...
            1:size(im,2))+1);
```

Figure 3. The histogram equalization code (left) and the vectorization (right).

```
/* The load function declares the type of
   pattern as one that operates on matrix
   multiplication taking the given
   dimensionalities. When the transformation
   needs to be applied, ''tform_func''
   will be called. */
PatternTransform load()
{
  Dimensionality lhs_dims[]={ Ri,'*'},
                 rhs_dims[]={'*', Ri},
                 out_dims[]={  1, Ri};
  return PatternTransform(OP_MATRIX_MULT,
                          lhs_dims,rhs_dims,
                          out_dims,tform_func);
}
/* The t-form replaces the parse tree node
   A*B with sum(A'.*B,1) */
Expression tform_func(BinaryExpression & be,
                      Identifier loopids[])
{
  be.setLHS(UnaryExpression(be.getLHS(),"'");
  be.setOperator(BinaryOperator(".*"));
  Expression  args[]={be,Constant(1)};
  return FunctionCall(Identifier("sum"),args);
}
```

Figure 2. Example of C++ style pseudocode for the functions that are defined in a DLL for Pattern ID=1.

age processing operation, namely *histogram equalization* (Fig. 3). Histogram equalization seeks to transform image intensities so that the histogram of the intensities becomes almost uniformly distributed. For an 8-bit image, a 1×256 lookup table `heq` is created and used to transform input image intensities to output intensities. The loop-based implementation iterates through the input image, `im`, mapping each pixel intensity through the lookup table and writing into the output image, `im2` (offsetting intensities by 1 to ensure indexing starts at 1 instead of 0). The more efficient way of performing this operation is to index the lookup table with the entire image at once, resulting in an output matrix with the same dimensions as the original image. As our dimensionality checking preserves the rules of Matlab, the double nested loop is replaced with a single statement. An 800x600 monochrome image (`im` has type `uint8`) takes 0.178 seconds for the loop version and roughly 0.114 seconds for the vectorized code, giving a speedup of roughly 1.56. Focusing on the loop portion of the code only, the original double nested loop takes 0.0814s and the vectorized statement takes 0.0176s giving a speedup of 4.6.

To demonstrate the potential of dimensionality analysis in a vectorizer, we present an illustrative example of the higher-level transformations that it may carry out. The original source code and the vectorized code are displayed in Figure 4. The example uses several non-pointwise transformations, including several of the transformations discussed in Section 3 and other transforms related to matrix multiplication. Notice that these transformations result in vector-

Settings	input time (s)	vect. time (s)	speedup
i=500 p=5000	0.536s	0.030s	~ 17
N=1000	0.174s	0.012s	~ 14
n=40	0.622s	0.0001s	~ 5000

Table 3. Timing results using stated settings for vectorization results on the Menon & Pingali examples given in Fig. 5

ized code that utilizes several Matlab built-in function calls, such as `repmat`, `size`, and `sum`. The vectorized code completes in approximately 0.5 seconds, whereas the original code took roughly 25 seconds (a speedup of roughly 50).

The vectorizer is also capable of proper vectorization of three of the four examples used in the work of Menon & Pingali [17] (see Fig. 5). Each of these examples contains an additive reduction statement. The only unsuccessful example is an optimization over order of evaluation of high-level operations and is an example that we see complementary to the task of vectorization. Timing results for the input source and the resulting vectorization on sample settings are given in Table 3. The speedup is dependent on the chosen problem size, but these results indicate the significant speedup possible on large problems or deeply nested loops (e.g., a speedup of 5000 on the third example for a moderate $n = 40$).

6. Related Work

Vectorization has its roots in loop parallelization, which splits up loops into parts that may be computed independently of each other for the purpose of sending them to separate processors in a multiprocessor architecture. The same idea can be applied to the vectorization of Matlab code: if many loop iterations can be done independently, a vector of the operands can be supplied to a vector operation instead. The foundation for a vectorizing compiler [2] and techniques to increase the amount of vectorization [19] now appears in textbooks [1, 22]. In particular, the dependence-based vectorizer of Allen & Kennedy served as the basis for our vectorizer. These techniques do not reduce nested loop statements to high-level matrix operations nor do they attempt to vectorize statements that are of the same dimensionality (a row vector cannot be added to a column vector). Van Beusekum acknowledged the potential for this problem in his vectorizer for Octave [21], but unlike our work nothing was done to ensure that a transpose was added or that only expressions with compatible dimensions were vectorized.

Our abstraction of dimensionality resembles the typed loop fusion introduced by Kennedy and K. S. McKinley [13] although the methods operate under different contexts. In typed fusion, loops are classified into two types — sequential and parallel — and only loops of the same type are fused. In our approach, the vectorized dimensionality can be seen as the type of an expression. Vectorization only occurs when the "types" of operands to an expression agree or when the types match some pattern and can be transformed into some resulting type (*i.e.*, the output dimensionality).

The vectorization described in this paper provides a strong argument for the automatic discovery of data types and array shapes — both expensively determined dynamically in Matlab. Olmos & Visser perform a source-to-source transformation of Octave code, substituting typed variables in place of untyped ones [18]. Their two-pass type inferencer first resolves data types and then determines array shapes. In contrast to this static solution, Chauhan & Kennedy pick up necessary information at runtime, enabling delegation to optimized low-level language subroutines rather than generic libraries [5]. Their transformations employ slice-hoisting, which finds the 'slice' of code determining the size of an array and 'hoists' this above its first use for runtime knowledge of the array size. Their approach obtains more accurate array-size information than could be available at compile time, which can result in significant gains in large matrix operations. McCosh finds sets of legal type configurations using propositional logic constructs and infers the size of the variables by formulating this problem as a special version of the clique problem which she solves in polynomial time [15]. Joisha *et al.* present a framework to infer array shape and even allow certain static inferences to be made when complete static knowledge is lacking, by establishing what useful inferences can be made regarding the shapes of statically indeterminable arrays [11, 10]. These techniques could be integrated with our vectorizer to deliver an end-to-end solution to the vectorization of Matlab programs.

The construction of a DDG and the correct ordering of vector statements to preserve data dependencies are the backbone of any vectorizer. Although additional improvements can be made in the construction of the DDG, another area for research is to exploit features of the source language. In Matlab, high-level matrix operations, such as matrix multiplication, are more efficient than their loop counterpart. This is primarily due to the large interpretive overhead of running loops sequentially [16]. The improvement to the baseline vectorizer results from its ability to detect complicated patterns and to convert them to their fastest-running high-level matrix equivalents. To achieve this goal, Menon & Pingali use an Abstract Matrix Form (AMF) as a language to express "numerical array objects" (matrices and loop nests) in a manner conducive to modification of the structure of these objects [17]. In their work, the Matlab

```
%!A(*,*) B(*,*) C(*,*) D(*,*) h(*) a(1,*) ind(1,*)
ind=1:750;
for i=2:2:1500,
  B(i,1)=D(i,i)*A(i,i)+C(i,:)*D(:,i);
  for j=3:2:1501,
    A(i,j)=B(i,ind)*C(ind,j)+D(j,i)'-a(2*i-1);
  end
end
```

```
ind=(1:750);
B(2*(1:750),1)=(D(2*(1:750)+(2*(1:750)-1)*size(D,1)).*A(2*(1:750)+...
              (2*(1:750)-1)*size(A,1)))'+...
              sum(C(2.*(1:750),:)'.*D(:,(2.*(1:750))))';
A(2*(1:750),2*(1:750)+1)=(B(2*(1:750),ind)*C(ind,2*(1:750)+1))+  ...
                         D(2*(1:750)+1,2*(1:750))' ...
                         -repmat(a(2.*(2*(1:750))-1)',1,size(1:750,2));
```

Figure 4. The original annotated source code (top) and resulting vectorized code (bottom) showing that several transformations were automatically applied.

source code is converted into constructs resembling first-order logic called Abstract Matrix Form (AMF), in which a number of axioms are used to perform transformations. Following the transformations, the AMF is then converted back into source code. Their method appears to be able to vectorize complicated loops into high-level matrix operations (similar to the examples presented in Sections 3 & 3.1), but unlike our approach it is not clear if their method can properly deal with irregular matrix accesses, such as the diagonal accesses discussed on Page 5. Our dimensionality checking approach provides a simpler way to perform a similar optimization, as well as providing a means for an end-user to modify the pattern set since this means updating the pattern database instead of modifying the solution core. They optimize the order of evaluation of high-level matrix operations, but we see this optimization as a complimentary to vectorization, one which can be performed as a separate pass.

Along with type/shape checking, it is well known that interpretive overhead is one of the primary efficiency issues in interpreted Matlab code [16]. Techniques for overcoming these overheads include compilation to other high-level languages [6], Just-in-time compilation [3], and parallelization [8], all of which are complementary to vectorization. The FALCON compiler, which compiles from Matlab to other languages such as Fortran, also detects patterns in Matlab code and allows for user interaction in applying the corresponding transformation [6]. The transformations do not appear to use patterns for vectorization, but rather for allowing an optimized library call to be substituted for a given type/shape of operand and high-level operation. Just-in-time compilation has recently been introduced into the Matlab interpretive environment and in some cases it produces dramatic improvements [14]. As the current JIT only operates on certain data-types, its effects would be complemented by the vectorization proposed in this paper. Additionally, any parallelization can be performed on the vector statements, for example by substituting high-level parallel libraries (*e.g.* [20]) for sequential ones. A similar argument holds for compilation. A strategy called telescoping languages, in development at Rice University, uses domain-specific libraries to allow the user to create high-performance programs [4]. This strategy targets Matlab by performing procedure vectorization, where a procedure called within a loop is replaced by a single call to an equivalent procedure accepting vector arguments. When the procedure in question requires values that change during looping, procedure strength reduction is applied to split the procedure into the parts which compute loop variant (these keep running sequentially in the loop) and invariant (these are taken out of the loop) values.

An idiom-recognition system is presented by Hiroyuki for the purpose of converting array accessing statements into loops for more efficient, and usually system-dependent, implementations [9]. In his work, he considered matrix multiplication, dot products, replication of a scalar or column into a matrix, and transposes. His use of transposes was to permit higher efficiency use of the hardware (which is possible in Fortran as opposed to Matlab) whereas we use it to ensure the vector code produced is semantically equivalent to the original source code. Our pattern matching strategy allows matching of any user-defined patterns, not simply a few Matlab-defined implementations. We additionally handle cases where Matlab variables can possibly

Input Source	Vectorized Result

`for k=1:p, for j=1:(i-1),`
`  X(i,k)=X(i,k)-L(i,j)*X(j,k);`
`end end`
	`X(i,1:p)=X(i,1:p)-L(i,1:i-1)*X(1:i-1,1:p);`

`for i=1:N,for j=1:N`
`  phi(k)=phi(k)+...`
`  a(i,j)*x_se(i)*f(j);`
`end end`
	`phi(k)=phi(k)+...`
	`  sum(a(1:N,1:N)'*...`
	`  x_se(1:N).*f(1:N),1);`

`for i=1:n,for j=1:n,for k=1:n,for l=1:n`
`  y(i)=y(i)+...`
`  x(j)*A(i,k)*...`
`  B(l,k)*C(l,j);`
`end end end end`
	`y(1:n)=y(1:n)+...`
	`  x(1:n)'*...`
	`  (A(1:n,1:n)*B(1:n,1:n)'*C(1:n,1:n))';`

Figure 5. Vectorization results of three examples used by Menon & Pingali [17].

be in different forms at runtime (one variable can become scalar, vector, matrix), since this is not explicitly written in Matlab code as it would be by Fortran's SPREAD operation. His conversions would recognize a pattern in several ways to allow for system-dependent optimizations, and this is something not done by our method as it specifically targets the Matlab language (which has the same constructs on any platform or hardware). Jouvelot and Dehbonei present generalized reductions, being the reduction of single loop operations into special hardware-supported operations [12]. In their work, a code scan produces a symbolic store for each loop, and this store is used to query a pattern database containing code replacements. As in our work, this symbolic representation provides pattern recognition, although ours occurs at a lower level (*e.g.*, our patterns are based upon dimensionality and operators).

7. Conclusions

This paper presented a dimensionality abstraction that is useful for a Matlab™ vectorizer. Simple dimensionality checking ensures that the resulting vectorized code is correct, while also performing any transposes of vectors/matrices to make dimensionalities agree. Additionally, this work demonstrated that simple pattern matching detects and vectorizes non-pointwise operations, such as matrix multiplications, and generate vector code for special matrix access patterns (e.g., diagonal accesses).

The extensible loop pattern database framework treats function calls in the same manner as matrix accesses. For the pointwise mathematical functions, such as `cos`, `sin`, `sqrt`, this interpretation is correct. For example, the statement `Y(i,j)=cos(X(i,j))` would be correctly vectorized as `Y(1:100,1:100)=cos(X(1:100,1:100))`. The investigation of dimensionality analysis for non-pointwise functions is left for future work, where correct vectorization may be possible by defining the input and output dimensionalities of the function (similar to the transformations in Section 3).

Acknowledgments

Funding for Neil Birkbeck and Jonathan Lévesque was provided by NSERC postgraduate scholarships. We also acknowledge the helpful comments provided by the anonymous reviewers.

A. Matlab Quick Reference

A matrix in Matlab may have any number of dimensions. By default both scalars and row vectors are treated as two dimensional objects, with respective sizes of 1×1 and $n\times1$ ($1\times n$ for column vectors). For completeness, Table 4 presents a brief overview of Matlab notations and operations used in this paper.

References

[1] ALLEN, J. R., AND KENNEDY, K. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[2] ALLEN, R., AND KENNEDY, K. Automatic translation of fortran programs to vector form. *ACM*

Operator	Description
`size(X,dim)`	Return the size of `X` in dimension `dim`.
`size(X)`	Return a row vector containing the length of each dimension of X.
`repmat(X,[r,c])`	Replicate `X` along the 1st dimension `r` times and replicated along the 2nd dimension `c` times.
`eye(m)`	Return the `m`×`m` identity matrix.
`ones(m,n)`	Return a matrix of dimension `m`×`n` with each entry set to 1.
`zeros(m,n)`	Return a matrix of dimension `m`×`n` with each entry set to 0.
`.*,.+,.-,./`	Perform the operation along each element. E.g., `A.*B` gives a matrix of size(A) with element `(i,j)` being `A(i,j)*B(i,j)`.
`start:inc:end`	Return a row vector that contains the entries from `start` to `end` going up in increments of `inc` (`inc` is optional and assumed to be 1).
`diag(X)`	If `X` is a matrix `diag(X)` returns the elements of the diagonal in a column vector. If `X` is a vector `diag(X)` returns a matrix with `X` on the diagonal and zeros everywhere else.
`A(:)`	The colon operator flattens the entries of `A` into a column (column major).
`A(i,:)`	Return row `i` of matrix `A`.
`A'`	Transpose matrix `A`.

Table 4. Description of Matlab operators.

Transactions on Programming Languages and Systems (TOPLAS) 9, 4 (1987), 491–542.

[3] ALMÁSI, G., AND PADUA, D. Majic: compiling matlab for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (New York, NY, USA, 2002), ACM Press, pp. 294–303.

[4] CHAUHAN, A., AND KENNEDY, K. Reducing and vectorizing procedures for telescoping languages. *Int. J. Parallel Program. 30*, 4 (2002), 291–315.

[5] CHAUHAN, A., AND KENNEDY, K. Slice-hoisting for array-size inference in matlab. In *Proceedings of the 16th International Workshop on Languages and Compilers for Computing (LCPC)* (2003), L. Rauchwerger, Ed., Springer-Verlag, pp. 495–508.

[6] DEROSE, L., GALLIVAN, K., GALLOPOULOS, E., MARSOLF, B., AND PADUA, D. Falcon: An environment for the development of scientific libraries and applications. In *In Proc. of the KBUP95: First international workshop on Knowledge-Based systems for the (re)Use of Program libraries, Sophia Antipolis, France* (1995).

[7] EATON, J. W. Gnu octave, 2006. http://www.octave.org.

[8] HALDAR, M., NAYAK, A., KANHERE, A., JOISHA, P., SHENOY, N., CHOUDHARY, A., AND BANERJEE, P. A library-based compiler to execute matlab programs on a heterogeneous platform. In *IEEE International Conference on Parallel and Distributed Computing Systems (PDCS)* (August 2002).

[9] HIROYUKI, S. Array form representation of idiom recognition system for numerical programs. In *APL '01: Proceedings of the 2001 conference on APL* (New York, NY, USA, 2001), ACM Press, pp. 87–98.

[10] JOISHA, P. G., AND BANERJEE, P. An algegraic array-shape interference system for MATLAB. *ACM Transactions on Programming Languages and Systems (TOPLAS) 28*, 5 (September 2006), 848–907.

[11] JOISHA, P. G., SHENOY, N., AND BANERJEE, P. Computing array shapes in matlab. In *LCPC* (2001), H. G. Dietz, Ed., vol. 2624 of *Lecture Notes in Computer Science*, Springer, pp. 395–410.

[12] JOUVELOT, P., AND DEHBONEI, B. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing* (New York, NY, USA, 1989), ACM Press, pp. 186–194.

[13] KENNEDY, K., AND MCKINLEY, K. S. Typed fusion with applications to parallel and sequential code generation. Tech. Rep. TR93-208, Rice University, August 1993.

[14] MATHWORKS. Accelerating matlab. *Matlab Digest* (September 2002).

[15] MCCOSH, C. Type-based specialization in a telescoping compiler for matlab. Master's thesis, Rice University, Houston, Texas, November 2002.

[16] MENON, V., AND PINGALI, K. A case for source-level transformations in matlab. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages* (New York, NY, USA, 1999), ACM Press, pp. 53–65.

[17] MENON, V., AND PINGALI, K. High-level semantic optimization of numerical codes. In *ICS '99: Proceedings of the 13th international conference on Supercomputing* (New York, NY, USA, 1999), ACM Press, pp. 434–443.

[18] OLMOS, K., AND VISSER, E. Turning dynamic typing into static typing by program specialization in a compiler front-end for octave. In *International Workshop on Source Code Analysis and Manipulation (SCAM)* (2003), IEEE Computer Society, pp. 141–150.

[19] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Commun. ACM 29*, 12 (1986), 1184–1201.

[20] SCALAPACK. The ScaLAPACK project, 2006. http://www.netlib.org/scalapack.

[21] VAN BEUSEKOM, R. A vectorizer for octave. Master's thesis, Universiteit Utrecht, Utrecht, The Netherlands, February 2005. INF/SRC-04-53.

[22] WOLFE, M. *High Performance Compilers for Parallel Computing*. Addison Wesley Longman, 1994.