Automatic Vectorization for MATLAB

Hanfeng Chen(✉), Alexander Krolik, Erick Lavoie, and Laurie Hendren

School of Computer Science, McGill University, Montréal, Canada
{hanfeng.chen,alexander.krolik,erick.lavoie}@mail.mcgill.ca,
hendren@cs.mcgill.ca

Abstract. Dynamic array-based languages such as MATLAB provide a wide range of built-in operations which can be efficiently applied to all elements of an array. Historically, MATLAB and Octave programmers have been advised to manually transform loops to equivalent "vectorized" computations in order to maximize performance. In this paper we present the techniques and tools to perform automatic vectorization, including handling for loops with calls to user-defined functions. We evaluate the technique on 9 benchmarks using two interpreters and two JIT-based platforms and show that automatic vectorization is extremely effective for the interpreters on most benchmarks, and moderately effective on some benchmarks in the JIT context.

Keywords: Vectorization · Promoted shape analysis · MATLAB · Elementwise functions · Vectorizing user-defined functions

1 Introduction

Vectorization is a mature field which has been studied for decades. However, there are new challenges and opportunities for using vectorization concepts to speed up array-based programming languages such as MATLAB [8]. The key insight is that many operations in MATLAB support both individual element operations, such as *op(a(i))*, as well as elementwise (vectorized) versions that apply *op* to all elements in an array using just one call, *op(a)*. When a call is made to a built-in operation over an entire array, the underlying implementation can then utilize highly tuned and parallelized libraries. For example, MathWorks began supporting multithreading on elementwise functions in MATLAB 7.4 (R2007a).[1] Thus, it becomes beneficial to replace loops that apply operations on individual elements with one or more vectorized statements, where the operations are now applied to entire vectors or arrays. Indeed, this is standard advice given to MATLAB and Octave [12] programmers as a way of hand optimizing their programs.[2,3]

[1] http://www.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multithreaded-computation.
[2] http://www.mathworks.com/help/matlab/matlab_prog/vectorization.html.
[3] http://wiki.octave.org/FAQ#Porting_programs_from_Matlab_to_Octave.

In this paper we present an approach and tool (Mc2Mc)[4] that automatically detects loops that can be vectorized and automatically produces output MATLAB code with vectorized instructions replacing the loops. In addition to handling loops with built-in MATLAB operations, we also allow loops which call user-defined functions by providing an analysis that determines if user-defined functions have the appropriate elementwise behaviour. Furthermore, we support if-conversion to allow even user-defined functions with conditionals.

We have implemented Mc2Mc based on the McLAB front-end and Tamer infrastructure [4,9], and have used our implementation to study 9 benchmarks on two interpreter-based systems and two JIT-based systems. In the interpreter cases, the automatic vectorizer led to very large speedups on some benchmarks and moderate speedups for others, with geometric mean speedups of 19.1x for Octave 4.0 and 7.65x for MATLAB 2013 (JIT off). However, with systems supporting JITs, such as MATLAB 2013a (1st gen JIT on) and MATLAB 2015b (2nd gen JIT which is always on), the effect of vectorization is mixed with geometric mean speedups of 1.02 and 0.77 respectively. There are still benchmarks which benefit from over 10x speedup, however other benchmarks have loops which are handled very effectively by the JIT, and vectorization can drastically hurt performance. Thus, it no longer makes sense for a MATLAB programmer to hand vectorize all of his/her code. However, our automatic vectorization system would allow a programmer or execution engine to try various strategies and identify those which benefit from vectorization.

The main contributions of this work are:

- We present a tool (Mc2Mc) that automatically transforms scalar MATLAB programs to equivalent vector form;
- We propose an interprocedural promoted shape analysis to determine if scalar code can be modified to vector form in loops and user-defined functions;
- We evaluate the performance of automatic vectorization on 9 benchmarks over 4 different execution engines.

In the rest of the paper we first provide more background about key features of MATLAB in Sect. 2. We then provide a description of our techniques with an overview of our approach in Sect. 3; a more detailed look at two key components, promoted shape propogation in Sect. 4 and our handling of user-defined functions in Sect. 5; and an outline the two final phases, data dependence analysis in Sect. 6 and the actual vectorization in Sect. 7. Finally we provide our experimental evaluation in Sect. 8, related work in Sect. 9 and conclude in Sect. 10.

2 Motivation and Background

MATLAB provides many features which enable vectorization. In this section, we provide an introduction to those features, and a motivating example for our vectorization transformation.

[4] https://github.com/sable/mc2mc.

Matrix Indexing: Matrix indexing provides a way to retrieve a collection of array elements with one operation. Figure 1(a) shows a `for` loop which accesses items of array `m` one at a time, and Fig. 1(b) shows the equivalent matrix indexing version, which accesses all the items at indices stored in `v`. Also note that in Fig. 1(b), the pre-allocation of array `r` is not needed.

```
m = rand(1,n);
r = zeros(1,4);
v = [3,7,6,4];
for i=1:4
   r(i) = m(v(i));
end
```

(a) One element at a time

```
m = rand(1,n);
v = [3,7,6,4];
r = m(v);
```

(b) All elements using matrix indexing

Fig. 1. A `for` loop and equivalent using matrix indexing

Colon Operator: The colon operator is mainly used to create vectors, subscript arrays and specify for iterations.[5]

Creating Vectors: The expression `j:k` generates a vector from `j` to `k` if `j` is less than `k`. With an additional parameter `i`, the expression `j:i:k` produces a vector from `j` to `k` with the stride `i`.

Subscripting Arrays: Matrix indexing can be introduced with the colon operator, such as `m(1:n)` where $n \leq length(m)$. Moreover, `m(:)` denotes all elements of `m`.

Iterating a for-loop: A simple for-loop header is `for i=1:n`, which indicates that the for body should be executed once for every value of `i` in `[1,n]`.

For Loop Vectorization: The main topic of this paper is automatic for loop vectorization. To motivate this, we provide a small example in Fig. 2. Function *foovec* is a vectorized version of *foo*. Both the original and vectorized versions call the same user-defined function *bar*. This may seem strange, since *foovec* is passing *bar* a vector, whereas the original *foo* was passing *bar* a scalar. However,

```
function foo(n)
    A=rand(1,n);
    B=zeros(1,n);
    C=zeros(1,n);
    for i=1:n
        B(i)=bar(A(i));
        C(i)=sqrt(i);
    end
end
```

```
function foovec(n)
    A=rand(1,n);
    B=bar(A);
    C=sqrt(1:n);
end
```

```
function [z] = bar(x)
    t=x+1;
    z=sin(x);
end
```

Fig. 2. Example vectorization

[5] http://www.mathworks.com/help/matlab/ref/colon.html?searchHighlight=colon.

because all of the statements in *bar* work on both scalars and vectors in the right way, *bar* can be called from the vector code, and *bar* will return a vector of values. One important part of our work is automaticially identifying such vectorizable user-defined functions.

To automatically vectorize the loop there are several things to check. Firstly, we must apply some standard dependence tests. Secondly, for each built-in function called from the loop body, such as *sqrt*, we must ensure that it has appropriate elementwise behaviour. Finally, for each user-defined function called from the body of the loop, such as *bar*, we must ensure that the body of the called function contains only vectorizable statements. Finally, the resulting vectorized program can be cleaned up, in this case the unneeded initialization of B is removed.

3 Overall Structure of the Vectorizer

We have implemented our approach in a tool, Mc2Mc, which given a MATLAB program, automatically identifies vectorizable sections and transforms scalar code to the equivalent vector form. An overview of the workflow of Mc2Mc can be found in Fig. 3.

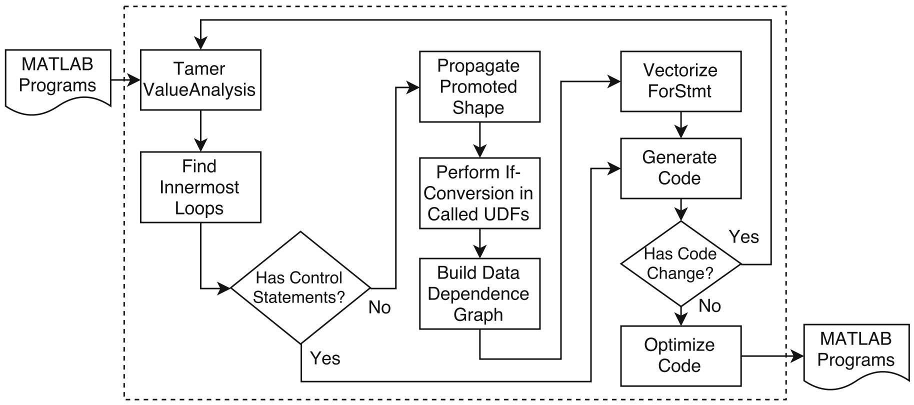

Fig. 3. The workflow of Mc2Mc.

The workflow begins by parsing input MATLAB programs into TameIR, a low-level representation used for analysis of MATLAB programs. The Tamer framework provides a set of interprocedural value analyses for shape information [6], as well as use-define and define-use chains that are used in later sections of the vectorizer. Once the initial analyses have been completed, the inner loops of the input program are collected for input to the vectorization algorithm. The vectorization algorithm thus uses an inside-out approach for handling loop nests [11]. Since the control flow of loops with nested if statements is not well suited for vectorization, only loops without such control statements are considered further.

Next, our new interprocedural promoted shape analysis is performed, determining whether scalar code can be correctly modified to an equivalent vector form. By propagating shapes through user-defined functions we expose additional areas for vectorization beyond built-in functions. While a user-defined function may not contain loops, nested if statements are permitted. A function is only considered vectorizable if all conditions and branches can be expressed in equivalent vector form. Promoted shape propagation is discussed in greater detail in Sect. 4 while user-defined functions and if-conversion are covered in Sect. 5.

Candidate statements for vectorization are then checked for dependencies that prevent vectorization. If no dependencies are found for a particular statement, the vectorized code can then be generated. Since the vectorization algorithm uses an inside-out approach, the pipeline may need to be rerun to handle nested loops and if statements in user-defined functions. Once a fixed point is reached, the newly vectorized code is statically optimized to remove unnecessary colon operators. In addition, dynamic checks are inserted to further reduce the performance impact of using the colon operators.

4 Promoted Shape Analysis

The promoted shape analysis is an interprocedural analysis that identifies the expressions in the body of a for loop that (1) have a scalar value that is derived directly or transitively from the loop index and that (2) can safely be promoted from a *scalar* form to a *vector* form, derived directly or transitively from the entire range of the loop index.

An expression (in a statement) can be promoted safely to a vector form if it performs the same operations on the values represented by variables that compose the expression over the entire range of the loop index.

In order to determine which expressions can be promoted safely, we perform a fixpoint analysis on the shape of variables. Initially, the loop index is first promoted from a scalar shape to a *promoted* shape, of the same shape as the entire loop index range. The promoted shape therefore represents a tentative replacement of the loop index variable scalar value with a vector that contains its entire range. All other variables shapes are initialized to *scalar*, *non-scalar*, or *unknown* ($\perp$) with the precise shape coming from the Tamer ValueAnalysis. The shape information is then propagated through every statement of the loop body and modified according to the effect of a statement's operation. The previous scalar shape of the output variables of a statement might be replaced with a promoted shape if the operation of a statement is compatible with the promoted shape of its input variable(s). If the operation is incompatible, the shape of the output variable will become $\top$.

Once the fixpoint is reached, all the statements that use a variable with a shape of $\top$ cannot be safely converted to vector form and therefore need to stay in the body of the for loop. All the others can potentially be moved outside of the loop body and the expressions that use variables with a promoted shape can be promoted to their vector form, as long as no dependency exist between the different statements (see Sect. 6).

In the remainder of this section, we first provide an explanation of the shape abstraction we use. We then explain which operations are compatible with a shape promotion. We finally provide the key parts of our promoted shape analysis in pseudocode.

4.1 Shape Abstractions

There are five abstractions summarized in Table 1. The initial variable with an unknown shape is denoted by $\perp$. The scalar shape S is considered because it can be extended in the context of elementwise operations. For the non-scalar N, it means the shape is neither a promoted shape nor a scalar. It is fine to have N in array indexing when the index is a scalar since the output of the array indexing is a scalar. For a promoted shape P, it is initialized by loop iterators and then propagated to variables. The $\top$ means there is no safe promoted shape for vectorization.

Table 1. Definitions of abstractions

Type	Description
$\perp$	An unknown shape
S	A scalar which is not promoted
N	A non-scalar which is not promoted
P	A promoted shape
$\top$	A shape cannot be vectorized

Note that a *promoted* shape represents a promotion from a scalar to a one-dimensional array. However some operations such as multidimensional array indexing (e.g. *A(i,i)*) may return a two-dimensional array when the i index variable is promoted (e.g. *A(1:n,1:n)*) rather than the diagonal of the matrix in the original loop. The expression is therefore not compatible with a promoted shape because it returns different values after the promotion. However, promotion along a single dimension (e.g. *A(i,j)* to *A(1:n,j)*) is possible if the shape of the array is compatible.

4.2 Compatible Operations

A unary function F satisfies the property of elementwise operations when it holds $\hat{R} = \overrightarrow{F}(\hat{A})$, where the $\overrightarrow{F}$ is a vectorizable function, the $\hat{A}$ denotes the promoted input parameter and the $\hat{R}$ denotes the promoted return value. A *promoted operation* is introduced in $A \rightarrow \hat{A}$ when a dimension in A is expanded to k_0, where $k_0 > 1$. That means $\hat{A}$ and A have the same number of dimensions, but $|\hat{A}| = k_0 \times |A|$, where the $|A|$ is its cardinality. Let $\rho(\hat{A})$ denote the new dimension (i.e. k_0). Let $\hat{A} = \{A_1, A_2, \ldots, A_n\}$ and $\hat{R} = \{R_1, R_2, \ldots, R_n\}$, where $\rho(\hat{A}) = \rho(\hat{R}) = n$, so that $\hat{R} = \overrightarrow{F}(\hat{A}) \Leftrightarrow [\{R_1, R_2, \ldots, R_n\}] = F(\{A_1, A_2, \ldots, A_n\})$.

A built-in function (BIF), which satisfies the property of elementwise operation, is vectorizable. For a unary built-in function F_u, it can be described as $\hat{R} = \overrightarrow{F_u}(\hat{A})$. However, a binary function F_b has three possible cases in vectorization. They are 1) $\hat{R} = \overrightarrow{F_b}(\hat{A}, B)$; 2) $\hat{R} = \overrightarrow{F_b}(A, \hat{B})$; and 3) $\hat{R} = \overrightarrow{F_b}(\hat{A}, \hat{B})$, where the A and B denote input arguments. It should be noted that the lengths of the argument A and B must agree in the third case.

User-defined functions are also compatible with input arguments in vector form under some conditions. An interprocedural sub-analysis, described in Sect. 5, is performed when a user-defined function is called from the body of a for loop to determine if the input arguments can indeed be promoted.

4.3 Key Parts of the Analysis

Initialization. The analysis starts from the innermost for loops. The variables in the body of the innermost for loops are initialized with one of the abstractions in Table 1. The loop index variables of all statements in the body of for loops are initialized to the promoted shape. All other variables are initialized to the scalar or non-scalar shape obtained from the Tamer ValueAnalysis. The pseudocode is provided in Algorithm 1.

Algorithm 1. Initialization

```
Data: a statement
Result: each variable with a promoted shape
1  foreach variable var in the statement do
2      if var.promotedShape has not been initialized then
3          if the statement is from a for-loop then
4              if var is the loop iterator then
5                  var.promotedShape ←from a scalar to a vector (i.e. loop's range);
6          else
7              if the shape of var is a scalar then
8                  var.promotedShape ←Scalar;
9              else
10                 var.promotedShape ←Non-scalar;
```

Promoted Shape Propagation in Statements. There are three important major cases for the propagation of the flow information, with the first case further sub-divided in three cases, as listed in Algorithm 2.

The first major case is a call to a function. A function call may target a built-in function or a user-defined function. For the BIFs, we separate them into two groups: elementwise built-in functions (eBIFs) and non-elementwise built-in functions (nBIFs). The eBIFs are compatible with a vector form under some conditions while most nBIFs are not. We therefore do not consider nBIFS and their return value is always $\top$. The rules for unary and binary eBIFs are defined in Tables 2 and 3 separately. In the Table 3, the N_d returns N if both have the same non-scalar promoted shape otherwise $\top$ and the P_d returns P if both have the same promoted shape otherwise $\top$. User-defined functions are covered in Sect. 5.

Table 2. The propagation rule for unary eBIFs

eBIF	$\bot$	S	N	P	$\top$
Output	$\bot$	S	N	P	$\top$

Table 3. The propagation rule for binary eBIFs

eBIF	$\bot$	S	N	P	$\top$
$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$
S	$\bot$	S	N	P	$\top$
N	$\bot$	N	N_d	$\top$	$\top$
P	$\top$	P	$\top$	P_d	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

The second major case concerns array indexing statements. For both *ArrayGetStmt* and *ArraySetStmt*, the promoted shape of the index variable needs to be the same as the shape of the array. Or the *ArraySetStmt* accepts a promoted shape `P` on the left-hand side and a promoted shape `S` on the right-hand side. In the *ArrayGetStmt* case, if so, the returned value's shape is set to *promoted*, otherwise it is set to $\top$.

The last major case, with the CopyStmt, trivially copies the shape of the left-hand side variable to the right-hand side variable.

4.4 An Example of Promoted Shape Analysis

To illustrate the promoted shape analysis, consider the loop from the function *needle* in the NW benchmark, as given in Fig. 4(a). In this example, *input_itemsets* is a matrix and *penalty* is a scalar. We would like to use our promoted shape analysis determine if the loop can be converted to vector form.

Our Mc2Mc tool first converts the code to a lower-level three-address style TameIR, as shown in Fig. 4(b). This means that each statement will now have at most one operation, which simplfies the subsequent analysis.

Figure 4(c) shows the result of the promoted shape analysis after each statement in the loop body. The loop iterator is used to get initial promoted shape. At program point 2, the `minus` is an eBIF which takes a promoted shape P (i.e. `i`) and a promoted shape S (i.e. 1). The output of the eBIF returns a promoted shape P for the variable `mc_t1`. Variable `mc_t1` and its promoted shape are then included in the flow set. The next statement has a unary BIF, `uminus`, which returns the same promoted shape as `mc_t1`. At program point 4, the variable *penalty* has a promoted shape S so that the eBIF `times` returns a promoted shape P. At program point 5, the array indexing on the left-hand side

Algorithm 2. Promoted shape propagation

```
1  PropagateStmt(assignStmt, inSet)
2      (lhs, rhs) ←assignStmt;
3      ps ←$\top$ ;
4      if the assignStmt is a CallStmt then
5          op ←rhs.getFunctionName();
6          args ←rhs.getArguments();
7          if op is a unary eBIF then
8              ps ←UnaryFunctionTable(op, args[1].ps);
9          else if op is a binary eBIF then
10             ps ←BInaryFunctionTable(op, args[1].ps, args[2].ps);
11         else if op is a UDF then
12             ps ←PropagateUDF(op, args, inSet);
13     else if the assignStmt is an ArrayGetStmt then
14         ps ←GetArrayIndexShape(rhs, lhs);
15     else if the assignStmt is an ArraySetStmt then
16         ps ←GetArrayIndexShape(lhs, rhs) ;
17     else if the assignStmt is a CopyStmt then
18         ps ←CopyPromotedShape(rhs.ps) ;
19     genSet(assignStmt) = {(lhs,ps)} ;
20     killSet(assignStmt) = {any tuple contains lhs};
21     outSet(assignStmt) = (inSet(assignStmt) - killSet(assignStmt)) $\cup$
       genSet(assignStmt);
22     return outSet
```

```
% penalty is a scalar
% input_itemsets is a matrix
for i = 2:max_rows
  input_itemsets(i,1)=-(i-1)*penalty;
end
```

(a) Original loop

```
[1]for i = (2 : max_rows);
[2] [mc_t1] = minus(i, 1);
[3] [mc_t2] = uminus(mc_t1);
[4] [mc_t3] = times(mc_t2, penalty);
[5] input_itemsets(i, 1) = mc_t3;
[6]end
```

(b) TameIR

```
[1] {(i,P)}
[2] {(i,P),(mc_t1,P)}
[3] {(i,P),(mc_t1,P),(mc_t2,P)}
[4] {(i,P),(mc_t1,P),(mc_t2,P),
     (mc_t3,P)}
[5] {(i,P),(mc_t1,P),(mc_t2,P),
     (mc_t3,P),(input_itemsets,N)}
```

(c) Flow sets

```
i=2:max_rows;
input_itemsets(i,1)=-(i-1).*penalty;
```

(d) Final vectorized code

Fig. 4. An example of promoted shape analysis

is a one-dimension promotion and the variable `mc_t3` has the same promoted shape. Therefore, the assignment is safe and the promoted shape of the variable `input_itemsets` is set to N. Finally, the analysis returns the set of promoted shape information. If a set of statements has no promoted shape $\top$ and there are no cyclic dependences, the statements can be vectorized safely. We then perform a final aggegration step on the TameIR, to produce back a MATLAB vectorized statement, as shown in Fig. 4(d).

5 Handling User Defined Functions

One of the key contributions of our approach is that we can vectorize loops which contain calls to user-defined functions (UDFs). The key insight is that if the body of the UDF contains only vectorizable statements, then the calling code can use the UDF as a vectorized operation. Since some UDFs contain conditional if statements, we have also developed a MATLAB-specific if-conversion to convert control dependence expressed as if statements into equivalent vectorized statements without control dependences.

5.1 Promoted Shape Analysis for UDFs

When the promoted shape analysis encounters a call to a UDF, the initial promoted shapes are propogated from the arguments of the call to the parameters of the called UDF. The promoted shape analysis is then used to propogate promoted shapes to all statements in the body of the UDF. At the end of the dataflow analysis, the return values are checked before they are copied back to the caller site. If any return value is neither a scalar nor a promoted shape, then the UDF is not vectorizable and all return values are set to $\top$ and then returned.

Since UDFs may include conditionals, we must extend the promoted shape analysis to handle conditional control flow. The key addition is that we apply the promoted shape analysis to each branch of the conditional, and then merge the results. More precisely, let *op2* be the function for binary eBIFs defined in Table 3, ps_1 and ps_2 are promoted shape from two different branches, and ps_{cond} is the promoted shapes of the condition of the if. The *merge* operator gets a new promoted shape with the following equation.

$$merge(ps_1, ps_2, ps_{cond}) = op2(op2(ps_{cond}, ps_1), op2(ps_{cond}, ps_2));$$

If a UDF is called multiple times from different caller sites, we follow a simple rule to solve the possible conflicting results from the analysis. The rule is that a UDF is kept the same no matter the changes in input arguments if the UDF is still vectorizable with the new arguments. Otherwise, the UDF is not vectorizable despite its prior result.

5.2 If-Conversion for UDFs

Some UDFs contain if statements, which would normally interfere with vectorization. However, there are some if statements which can be transformed into vectorized statements, using primitive vector operations available in MATLAB to combine results from the then and else branches.

Consider the example from the *CNDF* function of the *Blackscholes* (BS) benchmark, given in Fig. 5(a). The original code, with explicit control flow cannot be vectorized, because when *InputX* is promoted from a scalar to a vector, the if condition will execute only once instead of once per item in the vector. However, the computation can be converted to vector form as shown in Fig. 5(b). The trick is to create a boolean vector of 0's and 1's containing the results of the condition, and then to use this to select the appropriate values by multiplying by 1 for all values that should come from the then branch (and 0 otherwise). The same trick, with the negative conditions are used for the else branch. Then the two vectors are combined, giving all the results for both branches.

```
if InputX < 0
   InputX = - InputX;
   sign = 1;
else
   sign = 0;
end
```

(a) Original if-structure

```
thenCond = InputX < 0;
elseCond = not(thenCond);
InputX = thenCond.*(-InputX) + elseCond.*InputX;
sign = thenCond.*1 + elseCond.*0;
```

(b) After if-conversion

Fig. 5. If-conversion from the CNDF function of the BlackScholes (BS) benchmark

In general, if-conversion takes place when promoted shape can be safely propagated through the if-structure. Equivalently, the promoted shape must successfully propagate the new code after if-conversion. TameIR provides a simple if-structure with only then- and else-block. We first identify the variables which will be used in both the then- and else-block. We then analyze both branches using input flow. For variables which are used only in one block, there are two cases: (1) only used within block; or (2) remain after the if-block. The variables in (1) can be kept the same while the variables in (2) must multiply with its corresponding mask (i.e. *cond* or $\sim$*cond*).

6 Data Dependence Analysis

Besides promoted shape information, we consider the possible dependence between statements. It is the key problem for program vectorization. We investigate the exact test, the GCD test [1], to tell whether data dependence exists. If two statements cannot be decided by this test, we conservatively assume they have data dependence. Furthermore, a dependence graph is built on the result of the test. We split the graph into subgraphs in which each node connects but there is no connection between subgraphs. A subgraph is a directed graph. The Tarjan's algorithm [13] for finding strong connected components is adopted to identify possible acyclic subgraphs. Given an acyclic subgraph with no variable having promoted shape $\top$, we are able to get the topological order of each node in the subgraph with a topological sort. When vectorizing, the topological ordering is used to order the equivalent vector statements.

7 Vectorization and Optimization

The statements in a loop are separated into two groups: (1) vectorizable statements in a topological order and (2) non-vectorizable statements in a sequential order. For the first group, the loop range is extracted and each statement is vectorized and inserted above the loop. For the second group, the statements are not vectorizable and thus remain as is. If all statements are vectorizable, the resulting loop is empty and can be removed.

7.1 Special Cases

Function Replacement. MATLAB programs may contain many arithmetic operators, some of which can have different meanings depending on the operand types. Multiplication ($*$) for instance can either be an arithmetic or matrix multiplication. In MATLAB, a built-in function *mtimes* provides matrix multiplication while *times* performs an elementwise operation. With the Tamer Value-Analysis, we can generate improved code by using the faster elementwise function where possible. This replacement also applies to division (*mrdivide* vs. *rdivide*) and power (*mpower* vs. *power*).

Idioms for Reductions. MATLAB programs also commonly use patterns within loops, especially accumulation [2]. Using cycles from the dependence graph, common patterns can be replaced using the equivalent reduction operation. MATLAB provides a built-in reduction function *sum* for accumulation. The Mc2Mc tool is able to detect this idiom and generate vectorized code with the *sum* function.

Special Built-in Functions. Some built-in functions are excluded from the promoted shape analysis since they are not elementwise functions. However, they can be analyzed to expose further vectorization opportunities. We identify two such functions below.

Colon: Since we adopt an iterative method to vectorize loops from innermost to outermost, the generated code from a previous iteration may contain multiple calls to the colon operator. Since the colon operator is not elementwise, it is not included in the initial promoted shape analysis. To expose further vectorization, we give the return variable of a colon operator promoted shape N. This allows vectorization of outer loops which require full promoted shape information.

Transpose: Since the promoted shape of the function argument may be either a row or column vector and the vectorized function may require a particular shape to be semantically equivalent, we use the *transpose* built-in function to transform the inputs as needed.

7.2 Code Optimization with Dynamic Checks

Since indexing using a colon operator has an impact on the performance of vectorization, we explore dynamic checks to reduce the overhead caused by the redundant array indexing. If a colon indexing covers all elements in an array, the colon indexing can be replaced with an array name to improve performance. Only the left-hand side of an assignment statement is considered for the dynamic checks.

8 Evaluation

To study the performance of our automatic vectorization we have performed experiments on a diverse set of benchmarks on four different execution engines.

8.1 Experimental Setup

The experiments were done on a desktop with an i7-3820 3.60 GHz (eight cores) CPU and 8 GB RAM running Ubuntu 14.04 TLS. We selected four execution engines. We used two interpreters: Octave 4.0, which is an open-source interpreter and MATLAB 8.1 (R2013a) with the JIT turned off. We used two JIT-based systems: MATLAB 8.1 (R2013a) which has a 1st-generation JIT, and MATLAB 8.6 (R2015b) which has a newer 2nd-generation JIT. Each benchmark was executed 5 times and the mean execution time is reported. We used the Wu-Wei Benchmarking Toolkit to perform the experiments.[6] The source code of these experimemts is available on GitHub.[7]

There are total nine benchmarks chosen for the experiments, taken from the Ostrich benchmark set which provides multi-language versions of benchmarks covering a wide range of numerical categories (Dwarfs).[8]

Back-Propagation (BP): a method of training artificial neural networks. It provides an interactive algorithm to update the weights in the given network.

Black-Scholes (BS): a computationally intensive algorithm which is used to calculate the price for a portfolio of European options analytically with the Black-Scholes partial equation (PDE).

Capacitance (CAPR): computes the capacitance of a transmission line using finite difference and Gauss-Seidel iteration.

Crank-Nicholson (CRNI): computes the Crank-Nicholson solution to the one-dimensional heat equation.

Fast Fourier Transform (FFT): computes FFT on a random data set as input.

Monte-Carlo simulation (MC): approximates the value of π.

Needleman-Wunsch (NW): calculates optimal global alignment of two DNA sequences.

Page-Rank (PR): link analysis algorithm.

Sparse Matrix-Vector Multiplication (SPMV): compressed sparse row (CSR) format multiplication between a sparse matrix and a vector.

[6] https://github.com/Sable/wu-wei-benchmarking-toolkit/.
[7] https://github.com/Sable/lcpc16-analysis.
[8] https://github.com/Sable/Ostrich.

8.2 Experimental Results

To study the performance influence caused by the code vectorization, we compared the original MATLAB code with the automatically vectorized code. To produce the vectorized code we used our tool to identify and transform loops which could be vectorized, and we replaced the original loops with the automatically generated vector code.

Overall Performance. The results of our experiments are given in Table 4. There are four multicolumns, one for each execution engine. For each of these there are three columns: time for the original code, time for the automatically vectorized code, and speedup which is the ratio of *orig_time/vect_time*. We also provide the geometric mean speedup for each execution engine. A speedup of k means that the vectorized version was k times faster than the original loop version. In Table 4 we have shown all speedups ≥ 1 as bold blue numbers. For each benchmark (i.e. each row in the table) we show the time of fastest version over all the execution engines as bold italic red numbers.

Table 4. Times (in seconds) and Speedups (SU) (orig. time/vect. time)

	Octave 4.0 (interpreter)			MATLAB 2013a (interpreter)			MATLAB 2013a (1st gen JIT)			MATLAB 2015b (2nd gen JIT)		
Benchmark	orig. time	vect. time	vect. SU	orig. time	vect. time	vect. SU	orig. time	vect. time	vect. SU	orig. time	vect. time	vect. SU
BP	1855	***0.83***	**2235**	138.4	2.76	**50.1**	6.18	3.00	**2.06**	2.18	3.09	0.71
BS	97.1	0.20	**485.5**	28.84	0.14	**206**	4.84	0.13	**37.2**	1.35	***0.09***	**15.0**
CAPR	207.6	203.7	**1.02**	14.63	14.4	**1.02**	0.43	0.51	0.84	***0.23***	0.29	0.79
CRNI	2452	1075	**2.28**	248.7	119.7	**2.08**	7.67	40.9	0.19	***3.05***	3.66	0.83
FFT	80.88	76.2	**1.06**	12.95	13.0	**1.00**	3.83	7.05	0.54	***1.25***	2.13	0.59
NW	981.3	733	**1.34**	57.09	40.3	**1.42**	2.43	1.97	**1.23**	***1.12***	1.17	0.96
PR	511.3	5.00	**102.3**	49.75	1.95	**25.5**	1.28	1.16	**1.10**	***1.08***	1.15	0.94
MC	535.3	***0.35***	**1529**	128.3	0.59	**217.5**	3.75	0.55	**6.82**	0.93	0.46	**2.02**
SPMV	117.6	197.3	0.60	17.73	33.8	0.52	0.26	12.8	0.02	***0.20***	14.5	0.013
Geo Mean			**19.1**			**7.65**			**1.02**			0.77

The results are very interesting and show the relative importance of vectorization for different types of execution engines and show that although vectorization can lead to huge speedups, it is not always beneficial.

For the two interpreters we see excellent speedups due to vectorization. In the case of Octave we see speedups of 2235x for BP and 1529x for MC. In fact, these vectorized versions are the fastest overall, beating even the 2nd-generation JIT in MATLAB 2015b. The speedups for MATLAB 2013a (interpreter) are also quite impressive. However, the results also show that even with interpreters it is not always worth vectorizing, as illustrated by the slowdowns for SPMV. In this case the vectorized loop is the inner loop of the main computation, and the main compuation outer loop is not vectorizable. The inner loop executes on a vector of size 2, and thus is not a good candidate for vectorization.

In the case of the JIT execution engines, the results are more nuanced. Some benchmarks show only a small performance improvement, and others have small performance degradations. However, there still exist benchmarks where vectorization can give good speedups, namely BS and MC. Vectorization of BS gives 37.2x speedup for MATLAB 2013a (1st gen JIT) and 15x for MATLAB 2015b (2nd gen JIT). The reason is that the two benchmarks successfully achieve loop vectorization and UDF vectorization. The called UDFs are fully vectorized. The function invocations in BS are more complex than MC. Therefore, it is more difficult for the JIT to exploit possible parallelsim while our vectorizer achieves this. However, with the JITs there can be even more drastic performance degradations due to vectorization, as can be seen by the slowdown of SPMV. It would seem that vectorizing an inner loop that has very few iterations not only introduces overheads to that inner loop, but also likely interferes with the JIT's ability to generate efficient code for the entire loop nest.

9 Related Work

While vectorization is a mature field, there is no universal method for transforming scalar programs into vector form. Existing approaches either use user input, automated analyses or a combination of the two.

User-Guided Vectorization. Tian et al. implemented vector extensions to C and C++, allowing the Intel C++ compiler to produce efficient SIMD instructions without requiring low-level programming [14]. Using in-code directives, entire user-defined functions can be vectorized in addition to for loops yielding significant performance improvements. In constrast, our implementation allows vectorization of user-defined functions without code annotations. Klemm et al. also explored directive based vectorization by introducing non-vendor specific SIMD constructs to OpenMP [5]. Since not all loops can be automatically vectorized, experimental results show that using annotations improves performance over an existing production auto-vectorizing compiler. While evaluating the effectiveness of auto-vectorizing compilers, Maleki et al. also confirmed that production compilers can handle many synthetic benchmarks but have difficulty automatically vectorizing real world applications [7]. In our work, results show that auto-vectorization can still provide significant performance increases to substantial benchmarks, but that performance degredation is also possible, especially with modern MATLAB JITs.

A mixed user-automatic approach to vectorization has been implemented for MATLAB. Since vectorization of MATLAB code requires matrix sizes and shapes, Birkbeck et al. allow user shape annotations to guide the auto-vectorization techniques [2]. Additionally, a pattern based approach transforms common code patterns to the equivalent MATLAB built-in. Our implementation uses the same principles for vectorization, but can automatically infer the necessary shapes instead of using annotations.

Array Programming Languages. Array programming languages such as R and MATLAB are also important candidates for vectorization. Menon and Pingali showed that source-to-source transformations of MATLAB, including vectorization, can significantly improve program performance [10]. Vectorization allows better exploitation of the underlying hardware and reduces the interpreter overhead of repeatedly iterating the loop body. However, their exploration used hand-optimized programs and did not consider function vectorization as in our implementation. Chauhan and Kennedy introduced two optimizations: *procedure vectorization* and *procedure strength reduction*, which improved the performance of real digital signal processing applications [3]. The idea of *procedure vectorization* is similar to our approach to UDFs, replacing a function call inside a loop by a single function call with vectorized arguments. However, their transformation is achieved by hand while we present an automatic method for handling UDFs.

The R programming language provides a popular built-in function *lapply* which runs a given function on a list of input. By replacing the looping execution of *lapply* with a vectorized version of the supplied function, Wang et al. achieved meaningful speedups [15]. However, their implementation is both limited to *lapply* and can also generate inequivalent vector code from if statements due to the semantics of the R *ifelse* built-in function. Our work accepts more general input, and generates equivalent vector code when vectorizing if statements.

10 Conclusions and Future Work

We have presented an automated technique to detect and transform loops to vectorized code in MATLAB. Our approach introduces a new promoted shape propogation analysis which is used to identify vectorizable statements and user-defined functions.

We have implemented our approach as the Mc2Mc tool and used it to experiment with 9 diverse benchmarks over 4 different execution engines. From our experimental results we conclude that our automatic vectorizer can find and transform loops in a wide range of benchmarks. The vectorized code is usually faster, and sometimes three orders of magnitude faster, on interpreted engines. There is less benefit for vectorizing on JIT systems, but there still exist benchmarks where excellent speedups can be achieved by vectorizing. Our results also show that the general advice of "vectorize to improve performance" is not always true, especially in the JIT settings where vectorizing can interfere with the JIT.

In our future work we would like to integrate our automatic vectorizer into a MATLAB or Octave IDE, so programmers could selectively vectorize loops. We would also like investigate automatic and profile-driven techniques for deciding when vectorization is beneficial, and perhaps also develop some "unvectorizing" techniques for converting vectorized code to loops when vector code is deemed to be less efficient.

Acknowledgments. We would like to thank the McLAB group for providing the analysis framework, Tamer. This work was supported, in part, by NSERC.

References

1. Allen, R., Kennedy, K.: Automatic translation of Fortran programs to vector form. ACM Trans. Program. Lang. Syst. **9**(4), 491–542 (1987)
2. Birkbeck, N., Levesque, J., Amaral, J.N.: A dimension abstraction approach to vectorization in Matlab. In: CGO, pp. 115–130 (2007)
3. Chauhan, A., Kennedy, K.: Reducing and vectorizing procedures for telescoping languages. Int. J. Parallel Prog. **30**(4), 291–315 (2002)
4. Dubrau, A.W., Hendren, L.J.: Taming MATLAB. In: OOPSLA, pp. 503–522 (2012)
5. Klemm, M., Duran, A., Tian, X., Saito, H., Caballero, D., Martorell, X.: Extending OpenMP* with vector constructs for modern multicore SIMD architectures. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 59–72. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30961-8_5
6. Li, X., Hendren, L.J.: Mc2FOR: A tool for automatically translating MATLAB to FORTRAN 95. In: CSMR-WCRE, pp. 234–243 (2014)
7. Maleki, S., Gao, Y., Garzarán, M.J., Wong, T., Padua, D.A.: An evaluation of vectorizing compilers. In: PACT, pp. 372–382 (2011)
8. MathWorks: MATLAB. http://www.mathworks.com/
9. McLAB: The McLAB tools for compiling MATLAB (2016). http://www.sable.mcgill.ca/mclab/
10. Menon, V., Pingali, K.: A case for source-level transformations in MATLAB. In: DSL, pp. 53–65 (1999)
11. Muraoka, Y.: Parallelism exposure and exploitation in programs. Ph.D. thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. UMI(71–21189), February 1971
12. Octave: GNU Octave. https://www.gnu.org/software/octave/
13. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
14. Tian, X., Saito, H., Girkar, M., Preis, S., Kozhukhov, S., Cherkasov, A.G., Nelson, C., Panchenko, N., Geva, R.: Compiling C/C++ SIMD extensions for function and loop vectorizaion on multicore-simd processors. In: IPDPS, pp. 2349–2358 (2012)
15. Wang, H., Padua, D.A., Wu, P.: Vectorization of apply to reduce interpretation overhead of R. In: OOPSLA, pp. 400–415 (2015)