MATLAB Parallelization through Scalarization

Chun-Yu Shei∗ Adarsh Yoga Madhav Ramesh Arun Chauhan
School of Informatics and Computing, Indiana University, Bloomington, IN 47405, USA
{cshei,ayoga,mramesh,achauhan}@indiana.edu

Abstract

While the popularity of using high-level programming languages such as MATLAB for scientific and engineering applications continues to grow, its poor performance compared to traditional languages such as Fortran or C continues to impede its deployment in full-scale simulations and data analysis. Additionally, its poor memory performance limits its performance. To ameliorate performance, we have been developing a MATLAB and Octave compiler that improves performance of MATLAB code by performing type inference and using the resulting type information to remove common bottlenecks. We observe that unlike past results, scalarizing array statements, instead of vectorizing scalar statements, is more fruitful when compiling MATLAB to C or C++. Two important situations where such scalarization helps is in expressions containing array subscripts and sequences of related array statements. In both cases, it is possible to generate fused loops and replace array temporaries by scalars, thus reducing the memory bandwidth pressure. Additional array temporaries are obviated in the case of array subscripts. Further, starting with vectorized statements guarantees that the resulting loops can be parallelized, creating opportunities for a mix of thread-level and instruction-level parallelism as well as GPU execution. We have implemented this strategy in a MATLAB compiler that compiles portions of MATLAB to C++ or CUDA C. Evaluation results on a set of benchmarks selected from diverse domains shows speed improvements ranging from 1.5x to 16x on eight-core Intel Core 2 Duo machine.

Keywords MATLAB, parallelization, SSE, GPU

1. Introduction

Programmers’ productivity in high performance computing (HPC) has been widely accepted as an important issue that needs addressing. One possible way to approach this problem is to enable end users to program in high-level languages with acceptable limits on performance penalty. MATLAB has the potential to serve this purpose. It owes its continued popularity not only to its user-friendly interactive development environment, but also to its relatively high-level, scripting, language that lets mathematical operations be expressed with natural ease. Unfortunately, MATLAB also continues to suffer from performance problems, especially, when compared to more traditional languages popular among HPC users such as, C and Fortran. This often forces MATLAB users to either scale back their applications or seek expensive redevelopment solutions such as rewriting their code in C or Fortran.

To address MATLAB’s performance problem we have been developing a compiler for MATLAB that identifies critical portions of code and translates those into C++ or CUDA C. Unlike past efforts [3, 8, 19, 20] we do not attempt to compile entire functions. Instead, we leverage the extensive and highly tuned MATLAB libraries, which are often multi-threaded, and restrict our translation to those pieces of code that are likely to incur high overheads in MATLAB. The most obvious choice for translation to a lower-level language are scalar loops in MATLAB. However, such loops are rare in well-written MATLAB programs. Far more common are outer time-loops that contain array statements. Moreover, scalar loops in MATLAB often do not contain loop-carried dependencies and can be vectorized [9, 21, 25]. Instead, we focus on array statements that are found in abundance in typical MATLAB programs, since the language syntax encourages users to write array statements. These could even come from loops vectorized by a prior compiler pass.

We have identified two types of frequently occurring code segments in MATLAB programs that greatly benefit from compilation to C++ or CUDA C: array expressions involving non-scalar subscripts, and sequences of related array statements.

We have found that for any greater-than-linear time operation such as matrix multiply, it is worth performing a copy of non-contiguous array section(s) being operated on before performing the actual operation in order to maximize spatial locality, even though it increases memory traffic. This is in contrast to earlier findings in APL [1], which found the “drag-along and beating” approach beneficial in all cases. However, for linear time operations, by avoiding the creation of such temporary array sections, we can reduce the amount of memory traffic and dramatically increase performance. Furthermore, array subscripting can be parallelized to achieve even greater speedups in many cases.

While earlier studies have almost universally advised vectorization in order to amortize the cost of type disambiguation and dynamic dispatch [9, 20, 25], we have found that when compiling to C++, it is beneficial to scalarize array statements into C++ loops with the goal of parallelizing them. It is still desirable for input programs to be in vector form, however, since starting from vectorized statements guarantees that the resulting loops will be parallelizable. We study the performance impact of parallelizing such scalarized loops using multi-level parallelism on CPUs (thread-level parallelism using OpenMP, and instruction-level parallelism using Intel SSE2 extensions) and offloading the parallel loops on GPUs.

We have implemented a compiler that uses type inference to translate portions of a MATLAB function into C++ or CUDA C. The

∗ Contact author.
compiler uses a modified version of Allen and Kennedy’s typed fusion algorithm [2] to cluster compatible vector statements together that are subsequently scalarized into fused parallel loops. Our algorithm first clusters compatible vector statements and then generates fused scalarized loops directly, greatly simplifying the fusion algorithm. The clustering step can also be followed by generation of CUDA C code, instead of scalarized C++ code. We replace temporary arrays by privatizable scalars on the fused loops, whenever possible, and perform a post-processing step to minimize the number of temporaries, thus reducing register pressure, by coalescing variables with non-overlapping live ranges.

We evaluated our approach on a set of benchmarks selected from a diverse set of MATLAB applications. We achieved application speedups ranging from a factor of 1.5 to almost 17 on an eight-core Intel Core 2 Duo machine, compared to MATLAB running with eight threads.

The main contributions of this paper include:

1. Identification of array statements involving element-wise operations or statements with array subscripts (i.e., arrays used as subscripts) as major potential sources of performance improvement in MATLAB programs, by translating those into a lower-level compiled language, such as C++ or CUDA C.
2. A practical algorithm that simplifies Allen and Kennedy’s typed loop-fusion by working with array statements, instead of loops, and directly generates fused parallel loops with arrays replaced by privatizable scalars, whenever possible.
3. Implementation and evaluation of the algorithm on a set of MATLAB benchmarks on a multi-core machine and a GPGPU platform.

2. Background

MATLAB is a dynamically typed high-level language targeted towards numerical computation, with an emphasis on ease of use. MATLAB’s control-flow operators consist of the if/else conditional, for/while loops, and switch/case statements that are commonly seen in procedural languages. It supports heavily overloaded operators, which greatly simplify many operations for the user. For example, to solve the linear system $Ax = B$ in a low-level language such as C, a user would be required to either find an appropriate library to call and link their program with, or write a significant amount of code themselves. In MATLAB, the user simply needs to write $x = A \backslash B$, with the matrix $A$ and vector $B$ defined, and the overloaded \ operator checks the types of $A$ and $B$ and performs matrix left-division automatically.

Another feature that simplifies programming in MATLAB is support for array statements. Sub-sections of arrays can be taken easily by simply subscripting them directly (e.g. $v(1:4)$, to select the first four elements), and it is even possible to, for example, select every other array element with ease. Note that MATLAB uses 1-based array indexing. If $v$ is an array, then $v(1:2:end)$ is a vector containing every other element of $v$. Due to heavy operator overloading, operations can be performed on arrays and array sections just as easily as on scalar values.

Through the use of highly optimized libraries such as BLAS [6], MATLAB achieves very good performance on common operations such as dense matrix multiply. Additionally, a large collection of domain-specific libraries (toolboxes) are available, and data-parallelism and coarse-grained parallelism are supported though parallelized libraries, parfor loops, and an MPI interface.

However, despite the existence of a proprietary Just-in-Time (JIT) compiler, code that contains a significant amount of control-flow and scalar operations does not perform well, compared to equivalent C implementations. This is because the dynamic type disambiguation and dispatch overhead overwhelms the actual amount of computation when the operations largely involve scalars or very small arrays. This is especially true of loops with predominantly scalar computations in their bodies.

A second major source of inefficiency arises when subscript expressions contain references to arrays. Array subscripts are not only allowed, but are in fact common in MATLAB since they offer simple idioms to slice, reshape, and permute multi-dimensional arrays. For instance the expression $A(b)$ refers to reordered elements of $A$ if $b$ is a permutation of positive integers in the index range of $A$. On the other hand, if $b$ is a “logical” array then the expression denotes a selection of those elements of $A$ for which $b$ is true. Unfortunately, these are also idioms that the interpreter cannot efficiently handle. The interpreter seems\(^1\) to create a new temporary array whenever such an expression is encountered. For operations that only scan an array section once (or a small number of times) creating such intermediate copies can lead to large unnecessary overheads. Additionally, while the interpreter leverages data-parallel implementations of vector operations such as min and power, it still does not avoid the creation of temporary arrays when their arguments contain subscripts.

Section 4 describes our approach to address these issues, which is the main contribution of this paper. However, we start with an overview of our compiler and a discussion of some high-level design and implementation issues in the next section.

3. Compiler Architecture

Figure 1 shows an outline of our MATLAB compiler. The compiler is designed to perform source-to-source translation as well as translation to C++ or CUDA C. Note that we have omitted several details in the figure for the sake of clarity, which we describe next.

The compiler leverages the Octave front-end. Octave is a GNU-licensed open-source project that is designed to be compatible with MATLAB at the language-level [13]. This gives us a robust mechanism to parse MATLAB as well as Octave source. Our compiler is implemented in Ruby and makes use of our homegrown Ruby-embedded domain-specific language, called RubyWrite, for rewriting the abstract syntax trees (ASTs). A filter translates the AST coming out of the Octave parser into the RubyWrite format. Before performing any analysis on the AST, all expressions are first completely flattened. For example, an expression $a+b+c$ will get split into two by introducing a temporary variable $t$, as $t=a+b; t+c$. This mimics what the interpreter is likely to do and also exposes all the hidden temporary variables that must be created when the program is interpreted. More importantly, it greatly simplifies the process of type inference by creating explicit variable names, and accompanying types, for all the sub-expressions. Further, the process of expression flattening helps provide a more accurate estimate of a program’s memory footprint by exposing the hidden temporary arrays.

Our type inference strategy works in two steps. First, type variables are introduced for each program variable and MATLAB code is inserted for computing the type values. For example a statement $x=1$; will cause the code $\text{iType}_x=x\text{1}$; to be inserted before the statement, where $\text{iType}_x$ refers to the “intrinsic” (or base) type of the variable $x$, which in this case is integer (’i’') since $x$ gets its value directly from a constant that is an integer. Similarly, for a statement $x=a+b$; the intrinsic type of $x$ is computed by inserting the code $\text{iType}_x = 1\text{XF}\text{sum}(\text{iType}_a, \text{iType}_b)$, where $1\text{XF}\text{sum}$ is the type transfer function for intrinsic types for the $+$ operator. The second step is to do an aggressive partial evaluation of the program. For this purpose the compiler utilizes a separately running MATLAB (or Octave) interpreter. A vast majority of type values get completely statically evaluated through this pro-

---

\(^1\) Since MATLAB implementation is proprietary, this is our best guess based on indirect evidence such as execution time.
4. Approach

4.1 Problem

In order to understand the performance bottlenecks in MATLAB programs, we profiled a set of MATLAB applications and quickly discovered that a large number of applications spent substantial amounts of time—in some cases almost all their time—in what appeared to be trivial operations. Invariably, these operations involved subscripting arrays with other arrays or range expressions. A MATLAB range expression is of the form a:b:b that represents integers a through b in increments of b. Range expressions can be used to conveniently carve out rectangular array sections that need not be contiguous. Indirection arrays can be used to permute multi-dimensional arrays. Both of these are MATLAB idioms that occur frequently in applications.

Consider the code snippet in Figure 2 from a MATLAB translation of the sequential version of the well known NASMG benchmark [4]. In this code n is a three dimensional cubic array of size (say) 10, and c, d, and u are permutations of the range 1:10. f is a short array of four scalar values. The operation . * is an element-wise product, which translates to scaling in the above case since MATLAB does an implicit scalar expansion when it expects a vector where a scalar is given. The above code performs a simple computation, but creates 27 array temporaries that are simply permuted copies of n, and is the biggest bottleneck in NASMG. Translating just the above piece of code into C results in about 2x speed improvement on a Core 2 Duo processor. The speedup increases to 7x if the implicit loops that get generated around each individual operations are fused. Such loop fusion also allows the resulting loop nest to be parallelized, since the loop nest is guaranteed to not have any loop-carried dependencies.

Inner loops that perform mostly scalar operations are another common cause of performance problems in MATLAB programs. Many such loops are vectorizable, and thus can be handled by the compiler in their vectorized form. We assume that an earlier pass has already vectorized such loops. It is useful to work with array statements resulting from vectorizable loops, instead of the loops in their original form, since that allows array statements to be clustered with other array statements and simplifies the generation of fused parallel loops, as explained later.

4.2 Scalarization

Scalarization is the process of converting a vector (array) statement into a sequential loop. This is typically used to execute array statements on single cores. However, such scalarized loops are also good candidates for loop-level parallelism.

While earlier studies have almost universally advised vectorization in MATLAB [9, 21, 25], our key observation is that in many cases, the reverse is beneficial when compiling MATLAB to C++. At the same time, by only compiling those sections of code on which MATLAB performs poorly, we continue to leverage MATLAB’s highly-optimized libraries for operations that it does handle well, such as matrix multiplication and other specialized kernels, such as FFT.

Figure 3 shows the outline of the algorithm used in the compiler to identify scalarizable clusters of statements and directly generate fused parallel loops around them. The algorithm performs integrated scalarization, fusion, scalar replacement, and scalar privatization. It operates on one basic block of the code at a time. We assume that prior passes have already performed loop-invariant...
Algorithm: Scalarize, Fuse, and Parallelize

Input: function F in SSA form; dependence graph of F
Output: modified F; helper functions

for each basic block B in F do
  W ← statements involving element-wise array operations in B
  foreach statement w ∈ W do
    T_w ← array size involved in w (call this "type" of w)
  foreach t = "type" of statements in B, in decreasing order of types do
    D ← modified dependence graph returned by Allen and Kennedy’s typed-fusion algorithm on B with type t
    B' = empty block
    foreach d ∈ D in topological order do
      if d is a simple node then
        add d to B'
      else
        let s be the sequence of array statements corresponding to the compound node d
        let V be the variables defined within s and not used after s (local variables)
        let f be a new function name
        generate function f with body consisting of scalarized fused parallel loop surrounding the statements in s
        set all the upwardly exposed variable references (use before definition) not in V as input args to f
        set all the variables defined in s that are not in V as output args of f
        replace all arrays in V by scalars and mark them privatizable (scalar replacement)
        add call to generated function f to B'
    B = B'

Figure 3. Scalarization and generation of fused parallel loops with scalar replacement.

Figure 4. Translated code for a portion of NASMG benchmark.

4.3 Loop-level Parallelism

Loops resulting from the algorithm are amenable to parallelization. Since we start with vector operations, the resulting loop is inherently parallel. One exception is the presence of subscripted array references that require copies to be inserted in scalarized loops, or other special measures [26]. Since we handle subscripts by flattening them using temporaries, this does not pose any special problem. Subsequent scalar replacement is able to remove such temporaries for array subscripts. In principle, such loops are not parallel since they cause a loop-carried dependence. For instance, consider the statement a(2:n) = a(1:n-1), which gets translated to temp = a(1:n-1); a(2:n) = temp; A fused loop around these statements would have a loop-carried dependence. However, when these

\[ \text{for (int ParaMtmp2 = 0; ParaMtmp2 < size[2]; ParaMtmp2++)} \]
\[ \text{for (int ParaMtmp3 = 0; ParaMtmp3 < size[1]; ParaMtmp3++)} \]
\[ \text{for (int ParaMtmp4 = 0; ParaMtmp4 < size[0]; ParaMtmp4++)} \]
\[ \text{double PARAMtmp7$1 = n$0_val[int(c$1_val[ PARA_Mtmp4]-1)+int(c$1_val[ PARA_Mtmp3]-1)*n$0_dims[0]+int(c$1_val[ PARA_Mtmp2]-1)*n$0_dims[0]*n$0_dims[1]]; \]
\[ \text{double PARAMtmp8$1 = PARAMtmp6$1_val * PARAMtmp7$1;} \]
\[ \text{double PARAMtmp10$1 = n$0_val[int(c$1_val[ PARA_Mtmp4]-1)+int(c$1_val[ PARA_Mtmp3]-1)*n$0_dims[0]+int(u$1_val[ PARA_Mtmp2]-1)*n$0_dims[0]*n$0_dims[1]]; \]
\[ \text{double PARAMtmp62$1 = PARAMtmp60$1 + PARAMtmp61$1;} \]
\[ \text{double PARAMtmp63$1 = n$0_val[int(d$1_val[ PARA_Mtmp4]-1)+int(d$1_val[ PARA_Mtmp3]-1)*n$0_dims[0]+int(d$1_val[ PARA_Mtmp2]-1)*n$0_dims[0]*n$0_dims[1]];} \]
\[ \text{double PARAMtmp64$1 = PARAMtmp62$1 + PARAMtmp63$1;} \]
\[ \text{double PARAMtmp65$1 = PARAMtmp49$1_val * PARAMtmp64$1;} \]
\[ \text{m$1_val[ PARA_Mtmp4+ParaMtmp3*m$1_dims[0]+ParaMtmp2*m$1_dims[0]*m$1_dims[1]] = PARAMtmp48$1 + PARAMtmp65$1;} \]
effectively eliminated, which made it safe to parallelize the loop.

Another possible problem arises when arrays, subscripted with other arrays, occur on the left-hand side of an assignment. For example, a(b) = a; where a and b are both arrays of same size, say n. Unless b is a permutation of 1:n, the scalarized loop is non-parallel. MATLAB follows the semantics of a sequential loop in such cases. Thus, the last assignment to an element of a persists. To preserve the semantics, we do not attempt to parallelize such statements.

5.1 Applications

For GPU testing, we used the NVIDIA Tesla card C1060 (4 GB memory). Each test was run five times, and the average execution time taken.

GCC 4.5.1 with optimization flags -O2 -ftree-vectorize.

R2010b on an 8-core Intel Xeon X5365 (3 GHz, 8 GB DDR2 memory, 8 MB L2 cache) running 64-bit Gentoo Linux 2.6.32, unless indicated otherwise. C code was compiled with NVIDIA CUDA C compiler release 3.1, which can generate either C++ or CUDA C code. Experiments studied were run using MATLAB R2010b on an 8-core Intel Xeon X5365 (3 GHz, 8 GB DDR2 memory, 8 MB L2 cache) running 64-bit Gentoo Linux 2.6.32, unless indicated otherwise. C code was compiled with GCC 4.5.1 with optimization flags -O2 -ftree-vectorize.

NAS benchmark suite, 

NASMG

Shallow Water 1D

nBody3D

2

Note that MATLAB’s mex compiler does not seem to perform any optimizations. Instead, it primarily serves as a way to package MATLAB code into executable binaries that could interface with C code. Therefore, the running times obtained with mex are practically identical to those for the MATLAB interpreter.
MATLAB Parallelization through Scalarization

Although we are still consistently faster in all cases, the relative slowdown in the amount we are able to outperform MATLAB’s interpreter, as illustrated in Figure 5, shows a decrease moving from 2 to 4 threads. However, as the stride size to 2, then 8, shows a similar decrease in performance as observed in the vector benchmarks. Somewhat surprisingly, however, are the consistently good results we achieve in the random permutation case. This is most likely because for matrices, providing two randomly permuted subscripts such as $A(randperm(n), randperm(n))$ (randperm(n) returns a random permutation of the integers 1:n) has the effect of randomly permuting one dimension, then the other. As a result, by taking advantage of MATLAB’s column-major storage format and traversing column-by-column, the memory access patterns for each column are limited to the input column currently being processed, and the number of cache and TLB misses is limited.

Finally, we observe that while we are initially able to outperform MATLAB’s interpreter for the random permutation test case, our performance eventually drops to equal that of MATLAB’s. We believed this to be a result of TLB misses, since the input vector sizes approached 400 MB for the largest input sizes we tested, and randomly accessing elements of such large vectors could conceivably result in many TLB misses. To verify this, we used the Performance Application Programming Interface (PAPI) to access the processor’s hardware performance counters and read the actual number of TLB misses incurred during our test. We found that for an input vector size of 49 million elements, the contiguous access case results in approximately 2.5 million TLB misses, while the random permutation case results in nearly 100 million TLB misses. Such a 50-fold increase in TLB misses indicates that TLB misses becomes the dominant bottleneck in the random permutation case for large vectors, which even our C code could not overcome.

We performed similar microbenchmarks on two-dimensional arrays (matrices) of various sizes, with similar subscript access patterns, shown in the middle plot in Figure 8. As in the vector case, subcripting contiguous sections of the input matrices provides the largest speedup relative to MATLAB. As expected, increasing the stride size to 2, then 8, shows a similar decrease in performance as observed in the vector benchmarks. Somewhat surprisingly, however, are the consistently good results we achieve in the random permutation case. This is most likely because for matrices, providing two randomly permuted subscripts such as $A(randperm(n), randperm(n))$ (randperm(n) returns a random permutation of the integers 1:n) has the effect of randomly permuting one dimension, then the other. As a result, by taking advantage of MATLAB’s column-major storage format and traversing column-by-column, the memory access patterns for each column are limited to the input column currently being processed, and the number of cache and TLB misses is limited.

Finally, we also performed a series of vector microbenchmarks to analyze the gains from using thread-level parallelism with generated loops, shown in the rightmost plot in Figure 8. The version of MATLAB used here did not use multiple threads for vector additions and multiplications (the newer version does). These results indicate that the more CPU intensive an operation is, the greater the benefit of increasing the number of threads. Vector addition and multiplication do not show much speedup as the number of threads increases because they quickly saturate the available memory bandwidth and become bandwidth limited. In fact, vector addition shows a slowdown moving from 2 to 4 threads. However,
computationally expensive operations such as \( \sin \) and \( \cos \) show a near-linear speedup all the way to 8 threads. It is important to note that the MATLAB interpreter is in fact still leveraging its optimized libraries in the baseline results we compare our approach to—the speed difference can be attributed to what we believe is the MATLAB interpreter making a copy of subscripted arrays before performing the actual computation. This conclusion is derived from the observation that the CPU utilization graph shows a period of single-threaded activity followed by one of multi-threaded activity, whereas the code generated by our approach immediately shows multi-threaded behavior.

In summary, our evaluation shows that even with highly optimized and threaded libraries, significant performance gains are possible with MATLAB code with compiler analysis that is aware of more context than possible with byte-code JIT compilation. In particular, it is possible to leverage parallelism available on modern machines at multiple levels, including on GPU accelerators. Vector statements that abound in programs in an array language, such as MATLAB, provide a good opportunity for this. Moreover, extending existing compiler techniques to such array statements results in a simple and effective compiler algorithm to selectively translate portions of MATLAB to a lower-level compiled language, such as C++ or CUDA C. The algorithm not only exposes parallelism at multiple levels but also reduces the memory footprint of the original program, resulting in significant performance improvement of the overall program. However, the translation also needs to watch out for the pitfalls that might lead to too much data copying, nullifying any performance gains.

6. Related Work

One of the earliest attempts to optimize array subscripts was by Abrams in the context of APL, where he used a technique he called beating and dragging along [1]. The idea was to carry the new shape of a reshaped array, instead of creating a copy of the array, and translate any subsequent subscript expressions such that they would work with the original shape. The techniques presented in this paper go beyond reshaping and apply to subscripts that may create any arbitrary section of an array that is allowed by the language.

Some of the early attempts to translate MATLAB programs to a lower-level language have reported that translation to C++ is less effective than translation for Fortran 90 [16]. We still chose to target C++ since we believe that significant progress has been made in C++ compilers since the 1990’s and the use of C++ gives us convenient access to several advanced libraries, including those written for Octave [13], and other advanced numerical libraries such as the Matrix Template Library (MTL) [23]. Additionally, the use of C++ makes it possible to use trampoline functions, as explained in Section 4, to handle arbitrary subscripts while ensuring that the common case is optimized.

A key enabling technology in our compiler is automatic type inference. Type inference as a general topic is widely studied in programming languages theory, especially in the context of ML [18]. The earliest documented work on type inference in the context of MATLAB is by de Rose and Padua for their FALCON project [21]. They used a traditional bidirectional data-flow based technique to infer types. However, their inference was restricted to two-dimensional matrices and to standard MATLAB procedures. FALCON’s approach cannot handle recursive procedures or even be applied directly to the newer versions of MATLAB. Subsequently, the MaJIC Just-In-Time compiler by Almási and Padua built on FALCON and performed limited type inference [3]. Unfortunately, due to the non-availability of a working version of FALCON, we are unable to compare our findings directly with it. Nevertheless, our approach has drawn several lessons from FALCON, including the overall formulation of types.

Our type inference algorithm uses staging, which enables staging of any optimizations we perform, including optimizing subscripts and loops. The notion of optimizing a program in steps, as more information becomes available, has been used in a version of ML, called MetaML, where it is called multi-staging [24]. Since then, “staging” has been applied in several other contexts for gradual optimization of code as more information becomes available.

Type inference has also been used to optimize dynamic dispatch in the context of object-oriented languages [12, 15].

Belter et al. presented a technique to optimize linear algebra kernels specified in MATLAB syntax using loop fusion [5]. Our approach of forward substitution is reminiscent of their technique of inlining loops and fusing them aggressively. However, it differs in three important ways. First, we do not explicitly fuse loops, since forward substitution gives us an equivalent and simpler transformation, because our operations are vectorized to start with. Second, we focus only on linear (level-1, in BLAS terminology) operations since we would like to leverage the effort gone into developing optimized libraries for operations with greater computational complexity that are difficult to replicate automatically. Finally, we handle general array subscripts including those that refer to potentially non-contiguous array sections, which is not a concern in the study by Belter et al.

In addition to compiling to lower-level languages, source-level techniques for optimizing MATLAB have also been proposed [9, 14, 17]. However, the performance improvements with purely source-level approaches are usually somewhat lower than those achievable through translation to lower-level languages.
7. Conclusion and Future Work

In this paper we have motivated the need to identify and optimize two related and critical types of code sections in MATLAB, array expressions involving non-scalar subscripts and sequences of related array statements. By avoiding the creation of many unnecessary temporaries introduced by the interpreter, we greatly reduce memory pressure, which is a huge bottleneck on today’s machines. We chose to generate C++ loops to leverage several modern numerical libraries that are often written in C++. The choice of C++ also lets us generate simpler generic code for subscripted accesses, while aggressive inlining by modern C++ compilers eliminates the overheads in common cases.

We described an algorithm for partitioning and scalarizing array statements with integrated fusion, scalar replacement, and scalar privatization to directly generate parallel loops. Our algorithm leverages automatic type inference that we have implemented in our MATLAB compiler. We implemented the algorithm in our compiler that is capable of generating C++ as well as CUDA C code.

We evaluated our algorithm on a diverse set of benchmarks on programs that are often written in C++. The choice of C++ also lets us generate simpler generic code for subscripted accesses, while aggressive inlining by modern C++ compilers eliminates the overheads in common cases.

Future work, in progress, includes developing automatic techniques to avoid pitfalls arising out of too much data copying.

8. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0811703.

References