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.

Categories and Subject Descriptors D [*3*]: 4—compilers

General Terms vectorization, loop-fusion, scalarization

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 disambiuation 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.

[Copyright notice will appear here once 'preprint' option is removed.]

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 \ 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 `sin` 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 `iType_x='i';` to be inserted before the statement, where `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 `iType_x = IXF_sum(iType_a,iType_b);`, where `IXF_sum` is the *type transfer* function for intrinsic types for the `sum` (+) 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.

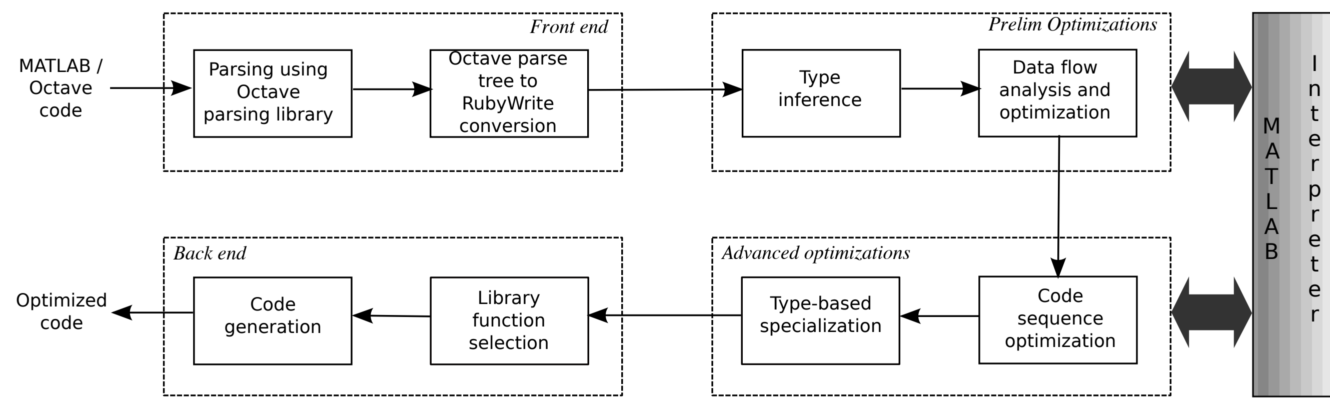

Figure 1. The overall architecture of the MATLAB compiler.

cess. Code for evaluating the rest of the type values is left behind to be evaluated at run-time that can be used to perform run-time (dynamic) optimization. A subsequent dead-code elimination pass gets rid of any type values that do not get used for run-time optimization. More details on our type inference approach are available [22]. Note that a similar method can be used to propagate any properties about variables that we care about, not just intrinsic types.

Before partial evaluation, the compiler converts the code into static single assignment (SSA) form [11]. The conversion from SSA follows standard textbook algorithms working on the control flow graph [10]. For converting out of SSA the compiler uses a recent algorithm that is more efficient and introduces fewer redundant copies than standard out-of-SSA algorithms [7].

The code generated by the compiler is a mix of MATLAB and C++ or CUDA C. Certain portions of the original MATLAB code are implemented as `mex` files, which are MATLAB's dynamically loadable compiled modules. We chose to generate C++ because that lets us use the open source C++ Octave library, if needed, and also keeps the option open to leverage modern generic libraries such as the Matrix Template Library [23].

4. Approach

4.1 Problem

```
m = f(1).*(n(c,c,c))
  + f(2).*(n(c,c,u)+n(c,c,d)
          +n(c,u,c)+n(c,d,c)
          +n(u,c,c)+n(d,c,c))
  + f(3).*(n(c,u,u)+n(c,u,d)
          +n(c,d,u)+n(c,d,d)
          +n(u,c,u)+n(u,c,d)
          +n(d,c,u)+n(d,c,d)
          +n(u,u,c)+n(u,d,c)
          +n(d,u,c)+n(d,d,c))
  + f(4).*(n(u,u,u)+n(u,u,d)
          +n(u,d,u)+n(u,d,d)
          +n(d,u,u)+n(d,u,d)
          +n(d,d,u)+n(d,d,d));
```

Figure 2. Code snippet from MATLAB version of `NASMG`.

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:s:b` that represents integers `a` through `b` in increments of `s`. 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) `N`, and `c`, `d`, and `u` are permutations of the range `1:N`. `f` is a short array of four scalar values. The operation `.*` is an elementwise 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

```
1  Algorithm: SCALARIZE, FUSE, AND PARALLELIZE
2  Input: function F in SSA form; dependence graph of F
3  Output: modified F; helper functions
------------------------------------------------------------
4  foreach basic block B in F do
5    W ← statements involving element-wise array operations in B
6    foreach statement w ∈ W do
7      T_w ← array size involved in w (call this "type" of w)
8    foreach t = "type" of statements in B, in decreasing order of types do
9      D ← modified dependence graph returned by Allen and Kennedy's typed-fusion algorithm on B with type t
10     B′ = empty block
11     foreach d ∈ D in topological order do
12       if d is a simple node then
13         add d to B′
14       else
15         let s be the sequence of array statements corresponding to the compound node d
16         let V be the variables defined within s and not used after s (local variables)
17         let f be a new function name
18         generate function f with body consisting of scalarized fused parallel loop surrounding the statements in s
19         set all the upwardly exposed variable references (use before definition) not in V as input args to f
20         set all the variables defined in s that are not in V as output args of f
21         replace all arrays in V by scalars and mark them privatizable (scalar replacement)
22         add call to generated function f to B′
23     B = B′
```

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

code motion so that redundant computation has been removed from loop bodies, and loop vectorization so that any parallelizable code is available as array statements. Both of these are standard compiler techniques [2]. Allen and Kennedy's typed loop fusion algorithm is used to construct maximal sequences of array statements. In our case the "loops" really are individual array statements. As a result, the pre-analysis that the algorithm requires to characterize each loop is considerably simplified. The "type" of an array statement is simply the size of the array(s) involved in that statement. Since we operate on statements that perform element-wise operation, all arrays involved in such a statement must have matching sizes.

The problem of determining an appropriate order in which to process the "types" in typed loop-fusion is NP-complete [2]. We use the heuristic of processing the types in the decreasing order of their values, i.e., array sizes. The rationale behind the heuristic is that preferentially generating fused loops around larger array sizes is likely to lead to bigger savings when temporary arrays are replaced by scalars in those loops.

Figure 4 shows abbreviated C++ code emitted by our compiler for the snippet of MATLAB code shown earlier in Figure 2

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

```
1  for(int ParaM_tmp2 = 0; ParaM_tmp2 < size[2];
       ParaM_tmp2++) {
     for(int ParaM_tmp3 = 0; ParaM_tmp3 < size[1];
         ParaM_tmp3++) {
3      for(int ParaM_tmp4 = 0; ParaM_tmp4 < size[0];
            ParaM_tmp4++) {
         double PARAM_tmp7$1 = n$0_val[int(c$1_val[
             ParaM_tmp4]-1)+int(c$1_val[ParaM_tmp3
             ]-1)*n$0_dims[0]+int(c$1_val[ParaM_tmp2
             ]-1)*n$0_dims[0]*n$0_dims[1]];
5        double PARAM_tmp8$1 = PARAM_tmp6$1_val *
             PARAM_tmp7$1;
         double PARAM_tmp10$1 = n$0_val[int(c$1_val[
             ParaM_tmp4]-1)+int(c$1_val[ParaM_tmp3
             ]-1)*n$0_dims[0]+int(u$1_val[ParaM_tmp2
             ]-1)*n$0_dims[0]*n$0_dims[1]];
7        ...
         double PARAM_tmp62$1 = PARAM_tmp60$1 +
             PARAM_tmp61$1;
9        double PARAM_tmp63$1 = n$0_val[int(d$1_val[
             ParaM_tmp4]-1)+int(d$1_val[ParaM_tmp3
             ]-1)*n$0_dims[0]+int(d$1_val[ParaM_tmp2
             ]-1)*n$0_dims[0]*n$0_dims[1]];
         double PARAM_tmp64$1 = PARAM_tmp62$1 +
             PARAM_tmp63$1;
11       double PARAM_tmp65$1 = PARAM_tmp49$1_val *
             PARAM_tmp64$1;
         m$1_val[ParaM_tmp4+ParaM_tmp3*m$1_dims[0]+
             ParaM_tmp2*m$1_dims[0]*m$1_dims[1]] =
             PARAM_tmp48$1 + PARAM_tmp65$1;
13     }
     }
15 }
```

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

statements are abstracted into a mex function after translation into C++, a is an input argument to that function. Since the mex functions follow copy-on-write semantics for their arguments, the loop-carried dependence is effectively eliminated, making it safe to parallelize the fused 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 not 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.

4.4 Instruction-level Parallelism

On modern x86 architecture, the SSE2 extensions allow aggressive Single Instruction Multiple Data (SIMD) parallelism to be exploited at the instruction level. Most modern compilers, including recent versions of gcc, are able to leverage these extensions. However, array references inside the loop body must obey certain constraints for this to happen. The C++ code generated by our compiler attempts to follow those constraints whenever possible. In particular, it keeps the array subscripts as simple as possible and avoids indirection, whenever possible. As a result, the C++ code emitted from our compiler is suitable for compiling using SSE2 extensions by a backend compiler.

4.5 Fine-grained Parallelism on GPUs

Our compiler can generate CUDA C code from the fused loops resulting from the algorithm in Figure 3. The compiler has a tunable upper limit on the number of threads to spawn on the GPU, by controlling the number of blocks and the number of threads per block. This guards against creating too many threads when processing large arrays. The compiler arranges for the input arguments to be first copied into the shared memory, through a device-to-device copy, and then performs computations from shared memory. This was found to be faster than operating directly from global memory. Allocation of privatizable scalars is controlled by the CUDA C compiler, but they are highly likely to be allocated in per-thread registers. Finally, an aggressive post-processing step tries to minimize scalar temporaries by merging names with non-overlapping live ranges in order to reduce the register pressure.

5. Experimental Evaluation

We have implemented our approach in our MATLAB compiler, which can generate either C++ or CUDA C code. Experiments 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'. Each test was run five times, and the average execution time taken. For GPU testing, we used the NVIDIA Tesla card C1060 (4 GB memory), compiled with NVIDIA CUDA C compiler release 3.1, V0.2.1221.

5.1 Applications

We evaluated our approach on five applications and one kernel written in MATLAB, drawn from diverse domains. The applications studied were nBody3D, NASMG, FDTD, Heated Plate, Forward and Shallow Water 1D. nBody3D performs a three-dimensional N-body simulation, NASMG is the multigrid benchmark from the NAS benchmark suite, FDTD applies the Finite Difference Time Domain technique on a hexahedral cavity with conducting walls, Heated Plate implements thermal simulation, Forward is a computationally intensive kernel within the Black Scholes application that analyzes stock market data, and Shallow Water 1D is a solver for shallow water equations in one dimension.

We were able to obtain speedups ranging from about 1.5 (nBody3D one thread) to almost 17 (NASMG eight threads) compared to MATLAB interpreter running in threaded mode, for three applications as Figure 5 shows[2].

In each case, we notice that the performance improvement flattens with increasing number of threads. We believe that this is because element-wise operations that these loops perform are linear-time operations, which rapidly saturate the memory bandwidth once more processing cores are involved. In the case of NASMG, a significant amount of time goes in computing temporary values due to its complex subscript expressions, as was shown in Figure 2. As a result, there may be slightly lower memory pressure, leading to somewhat better scalability with number of threads. However, the improvement is marginal beyond four threads.

One of the computation fragments in nBody3D shows 1.5x speed improvement when SSE is enabled. Figure 6 shows the impact of using SSE2 extensions on the code fragment shown in the figure. This code fragment is picked as an array statement sequence to be compiled into a mex function by our compiler However, its overall impact on the whole application is minimal as the graph for nBody3D suggests. In the case of NASMG and Heated Plate, indirect array accesses prevent the backend compiler from using SSE. Therefore, the graph for those do not include separate plots with SSE. The graph for NASMG includes plots for two different benchmark sizes, classes A and B, respectively.

Our evaluation on the remaining three benchmarks demonstrated that applying our scalarizing method indiscriminately can lead to performance degradation, instead of improvement. The table in Figure 7 shows the typical performance *slowdowns* with three of the applications on the CPU and two of the applications on the GPU. In each case, the data movement costs overwhelm any gains in speed due to compilation into C/C++ and parallelization. For the CPU case, the need for data copy arises out of the copy-on-write semantics of arrary arguments to mex calls. For GPU, the data needs to be moved back and forth between CPU and GPU memory. We are currently developing an algorithm that takes the data movement costs into account to filter out the candidate code segments that would require too much data copying. Our initial experience shows that each of the cases listed in Figure 7 can be filtered out using that method, thus avoiding the drastic slowdowns that might occur with a naïve approach.

5.2 Microbenchmarks for Memory Hierarchy Performance

To isolate the impact of memory hierarchy, we performed some microbenchmarks on vectors and matrices of various sizes while specifying different subscript access patterns. Figure 8 summarizes the results from our experiments. These tests were conducted using MATLAB version R2010a and gcc version 4.3.4.

The leftmost plot shows the speedups obtained on a single-dimensional array (vector) for increasing sizes, using eight threads on an 8-core machine for both C++ and MATLAB runs.

We first selected a contiguous portion of a vector from the beginning, and added it to a similarly selected contiguous portion of another vector. We then performed a similar test, only selecting contiguous portions from the ends of the vectors instead of the beginnings. We then tested strided accesses, with step sizes of 2 (selecting every other element) and 8. We selected a step size of 8 to be

[2] Note that MATLAB's mcc 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 mcc are practically identical to those for the MATLAB interpreter.

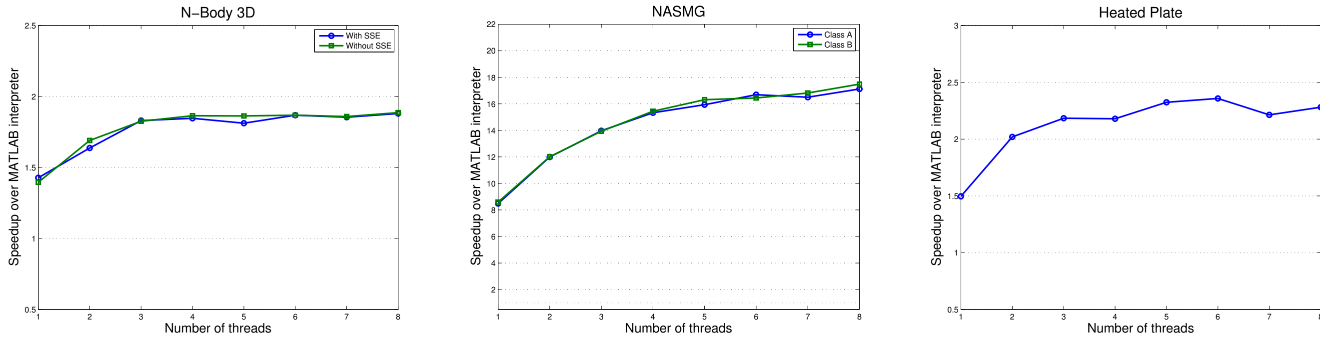

Figure 5. Speedups of a selection of benchmarks with varying number of threads, using OpenMP.

```
dr(:, :, 1) = dr(:, :, 1)./r;
dr(:, :, 2) = dr(:, :, 2)./r;
dr(:, :, 3) = dr(:, :, 3)./r;
```

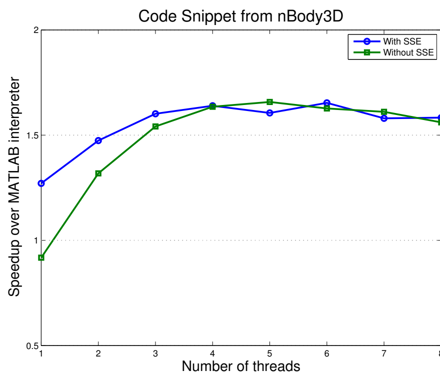

Figure 6. Snippet of code from `nBody3D` illustrating the impact of enabling SSE2 instructions.

Application	Naïve slowdown	
	One CPU thread	GPU
FDTD	3.27	
Forward	6.12	36.22
Shallow Water 1D	4.68	
nBody3D		2.18

Figure 7. Slowdowns with naïve application of the scalarizing algorithm.

included because it is the first stride size that doesn't benefit from spatial locality, as our array element sizes were 8 byte `doubles` and our test machine's cache line size was 64 bytes. Finally, to test worst case performance, we selected random permutations of the two input vectors to sum.

The two cases with contiguous subsections of vectors being selected behave nearly identically, despite the fact that we observed that in MATLAB's command window, accessing `A(1:(end-1))` takes approximately twice as long as accessing `A(1:end)` and `A(2:end)` for large vectors. Since MATLAB's internal implementation is proprietary, we do not have an explanation for this seemingly anomalous behavior.

As we increase the stride size to 2, then 8, we see a decrease in the amount we are able to outperform MATLAB's interpreter, although we are still consistently faster in all cases. The relative decrease in performance is most likely due to the reduced spatial locality such strided access patterns incur. Of course, MATLAB itself will also experience a similar reduction, but its extra interpretive overheads likely mask the effect, compared to our generated C code.

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, subscripting 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,

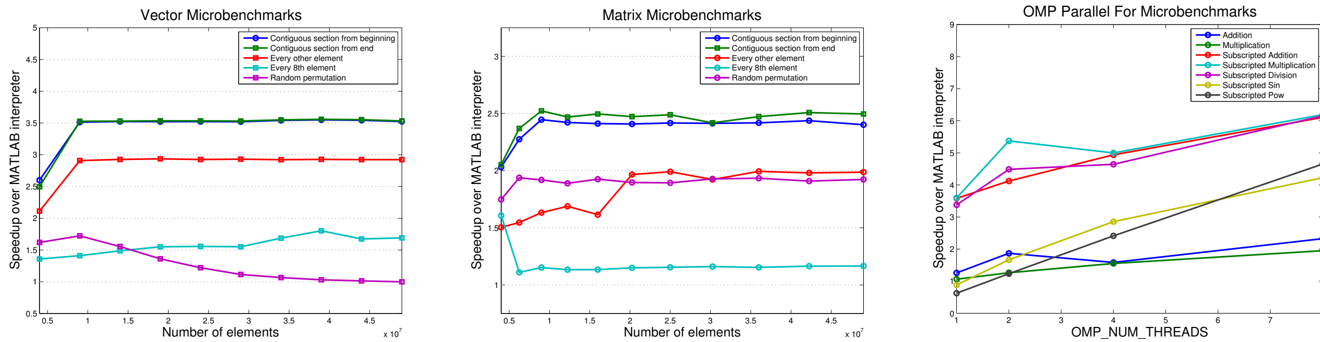

Figure 8. Speedups of array microbenchmarks with varying data access patterns and various operations.

computationally expensive operations such as `sin` and `pow` 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 multi-core machine as well as a GPU card.

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

[1] Philip S. Abrams. *An APL Machine*. Doctoral dissertation, Stanford University, Stanford Linear Accelerator Center, Stanford, California, USA, February 1970.

[2] John R. Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2001.

[3] George Almási and David Padua. MaJIC: Compiling MATLAB for Speed and Responsiveness. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.

[4] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Russell L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, Horst D. Simon, V. Venkatakrishnan, and Sisira K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[5] Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the Generation of Composed Linear Algebra Kernels. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[6] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hamarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software (TOMS)*, 28(2):135–151, June 2002.

[7] Benot Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2009.

[8] Arun Chauhan. *Telescoping MATLAB for DSP Applications*. Doctoral dissertation, Rice University, Department of Computer Science, Houston, Texas, December 2003.

[9] Arun Chauhan and Ken Kennedy. Reducing and Vectorizing Procedures for Telescoping Languages. *International Journal of Parallel Programming*, 30(4):291–315, August 2002.

[10] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, December 2003.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991.

[12] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 93–102, 1995.

[13] John W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002.

[14] Daniel Elphick, Michael Leuschel, and Simon Cox. Partial Evaluation of MATLAB. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830/2003, pages 344–363. Springer Verlag, 2003. DOI `http://dx.doi.org/10.1007/978-3-540-39815-8_21` not functional.

[15] Urs Hölzle and David Ungar. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):355–400, July 1996.

[16] Bret Andrew Marsolf. *Techniques for the Interactive Development of Numerical Linear Algebra Libraries for Scientific Computation*. Doctoral dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1997.

[17] Vijay Menon and Keshav Pingali. A Case for Source-Level Transformations in MATLAB. *ACM SIGPLAN Notices*, 35(1):53–65, January 2000.

[18] François Pottier and Didier Rémy. The essence of ML typing. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–490. The MIT Press, Cambridge, Massachusetts, 2005.

[19] Shankar Ramaswamy, Eugene W. Hodges IV, and Prithviraj Banerjee. Compiling MATLAB Programs to ScaLAPACK: Exploiting Task and Data Parallelism. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 613–619, 1996.

[20] Luiz Antônio De Rose. *Compiler Techniques for MATLAB Programs*. Doctoral dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana, Illinois, USA, 1996.

[21] Luiz De Rose and David Padua. Techniques for the Translation of MATLAB Programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, March 1999.

[22] Chun-Yu Shei, Arun Chauhan, and Sidney Shaw. Compile-time Disambiguation of MATLAB Types through Concrete Interpretation with Automatic Run-time Fallback. In *Proceedings of the 16th annual IEEE International Conference on High Performance Computing (HiPC)*, 2009.

[23] Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: Generic Components for High-Performance Scientific Computing. *Computing in Science and Engineering*, 1(6):70–78, November 1999.

[24] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. Doctoral dissertation, Oregon Graduate Institute of Science and Technology, Department of Computer Science and Engineering, Portland, Oregon, USA, November 1999.

[25] Remko van Beusekom. A Vectorizer for Octave. Masters thesis, technical report number INF/SRC_04_53, Utrecht University, Center for Software Technology, Institute of Information and Computing Sciences, Utrecht, The Netherlands, February 2005.

[26] Yuan Zhao and Ken Kennedy. Scalarization Using Loop Alignment and Loop Skewing. *The Journal of Supercomputing*, 31(1):5–46, January 2005.