High-Level Semantic Optimization of Numerical Codes

Vijay Menon and Keshav Pingali
Department of Computer Science,
Cornell University, Ithaca, NY 14853.
{vsm,pingali}@cs.cornell.edu

Abstract

This paper presents a mathematical framework to exploit the semantic properties of matrix operations in loop-based numerical codes. The heart of this framework is an algebraic language called the *Abstract Matrix Form* which a compiler can use to reason about matrix computations in terms of loop nests, high-level matrix operations, and intermediate forms. We demonstrate how this framework may be used to detect and exploit matrix products in loop-based languages such as FORTRAN and MATLAB, and discuss the resulting performance benefits.

1 Introduction

Algebraic properties of scalar integer and floating point operations are used by most compilers to optimize programs. These properties enable compilers to reduce of the strength of expressions, enhance the power of common subexpression elimination, and verify the legality of certain loop transformations [2]. Although matrices are also endowed with a rich algebra, it is less common for compilers to exploit matrix algebra to optimize programs. The major obstacle is that matrix operations are difficult to detect when they are hidden within loop nests and array subscripts. Once high-level matrix operations are exposed, there are many ways to exploit them to enhance program efficiency.

1. Efficient hand-tuned implementations are available for common high-level matrix operations. These are usually superior to compiler generated code. Perhaps the most relevant examples are the Basic Linear Algebra Subroutines (BLAS) [9]. Carefully hand-tuned versions of the BLAS library are ubiquitous on high performance architectures. Although core BLAS routines such as matrix-vector product (DGEMV) and matrix-matrix product (DGEMM) are exactly the type of codes for which compiler technology is most advanced (perfectly nested loops), there still remains a gap in performance between compiler and hand optimized versions of these codes. Figure 1 illustrates this margin on an IBM Power2 processor between hand optimized BLAS routines in the ESSL library [1] and code generated by the IBM XL FORTRAN compiler given the loop nests in Figure 2. [1] Over a variety of matrix sizes, both BLAS operations maintain roughly a 20% performance advantage over their compiler generated counterparts.

2. In an interpreted language such as MATLAB, utilizing high-level operations is vital for performance. All MATLAB operations incur overheads such as memory operations, type checks, and array bounds checks, but loops can tremendously magnify this overhead by performing redundant checks in each iteration. The most effective way to avoid loop overhead is to simply avoid the use of loops! In a high-level matrix operation, runtime checks are only performed once, at the beginning. Transforming MATLAB loops into higher-level operations provides a much more efficient way of performing the same computation, as we discuss later in the paper.

3. Once we are able to detect high-level operations, we can use their semantic properties to realize optimizations that are otherwise not feasible. For example, consider the MATLAB expression $A * B * x$ where A and B are $n \times n$ matrices and x is a $n \times 1$ column vector. The MATLAB interpreter, by default, evaluates expressions left to right; for this expression it will compute an $O(n^3)$ matrix-matrix product followed by an $O(n^2)$ matrix-vector product. However, the associativity of matrix products permits any order of evaluation. In this case, it is clearly more beneficial to evaluate from right to left, computing only two $O(n^2)$ matrix-vector products. By utilizing a high-level semantic property of matrix products, we obtain an asymptotic gain in performance.

In this paper, we present a framework for detecting point-wise matrix operations and matrix products in loop-based languages like MATLAB and FORTRAN, and show how the properties of these operations can be used to optimize programs by source-level transformation. This framework uses a language called *Abstract Matrix Form* (AMF) which is a convenient medium for

[0] This work was supported by NSF grants CCR-9720211, EIA-9726388 and ACI-9870687.

[1] This comparison was performed on a 120 MHz IBM Power2 Super Chip (p2sc) with a 128KB data cache and 256 MB of memory. All code was compiled with `-O3` and `-qhot` options.

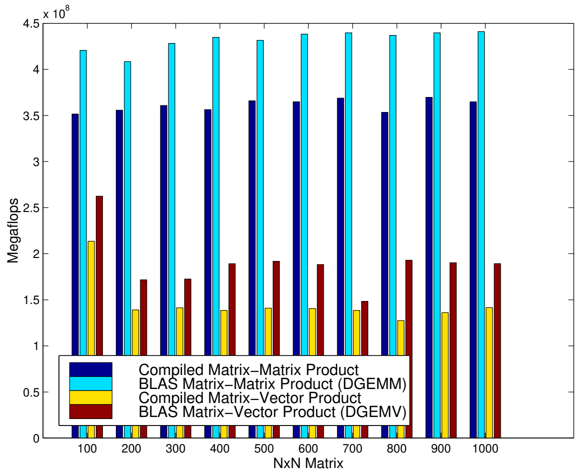

Figure 1: BLAS vs. Compiled code

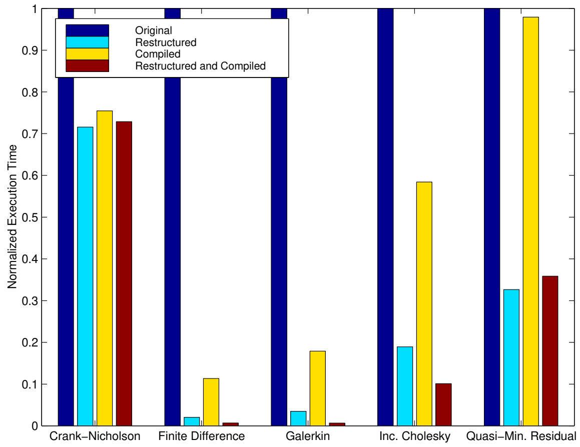

Figure 3: High-Level Optimization of MATLAB Programs

```
do i = 1, n
  do j = 1, n
    y(i) = y(i) + A(i,j)*x(j)
  enddo
enddo
```

Matrix-Vector Product

```
do i = 1, n
  do j = 1, n
    do k = 1, n
      C(i,j) = C(i,j) + A(i,k)*B(k,j)
    enddo
  enddo
enddo
```

Matrix-Matrix Product

Figure 2: Standard loop-nest representations of matrix products

performing source-level transformations. The need for such an intermediate language is not obvious since it seems at first sight that a language of matrix expressions, transformed by standard matrix identities, should be adequate for expressing optimizations. However, such a language is not convenient when programs contain a mixture of loops and high-level matrix operations, as is often the case in both MATLAB and FORTRAN-90 programs.

It may also seem that such a framework is unnecessary if the programmer codes directly in a matrix language like MATLAB or FORTRAN-90. To demonstrate that this is not the case, we performed a study of the Falcon benchmark suite of MATLAB programs [6] from the University of Illinois. It revealed that the transformations we propose are applicable to five of the twelve benchmarks in this suite. Figure 3 illustrates the effect of these transformations. The first two sets of bars denote the interpreted execution times of the original and transformed MATLAB codes (the interpreted execution time is normalized to 1.0). The last two bars in each set denote the execution time of the original and transformed codes when compiled by the MathWorks MCC MATLAB compiler. These results illustrate two surprising points. First, in all but one case, simple restructuring of MATLAB code achieves substantially greater performance gain when compared to MCC compilation! Second, in three of the benchmarks, the performance gains due to restructuring are complementary to those due to compilation; i.e., high-level restructuring and compilation combine to produce much faster code than either alone. These results suggest that source-level code transformations are important regardless of whether code is executed by interpretation or by compilation.

We present a more detailed analysis of these performance results in [18]. Our focus in this paper is on the compilation techniques to realize these benefits. The rest of this paper is organized as follows. First, in Section 2, we discuss background work relevant to this paper. In Section 3, we give four motivating examples, including two from the Falcon benchmark suite. In Section 4, we introduce our framework and define our abstract language. In Section 5, we demonstrate how a compiler may use this framework to realize optimizations. Finally, we conclude with ongoing directions of research.

2 Background Work

Algebraic properties of scalar arithmetic are used extensively in compiler optimizations. These properties are essential to common optimizations like constant folding and reduction of strength [2]. They may also be utilized to improve optimizations like common subexpression elimination. Commutativity and associativity of floating-point addition and multiplication are often essential to the legality of loop transformations such as tiling to enhance locality [23, 24]. Algebraic properties are also beneficial in obtaining more precise dependence information, particularly in the context of automatic parallelization [12].

The classical example of detecting and exploiting high-level operations in numerical codes is vectorization for vector supercomputers [3, 15, 20, 21, 25]. Vectorizing compilers for languages like FORTRAN reorder and isolate statements (mostly through loop distribution) that are then converted to vector operations. However, most of this work focuses on point-wise assignments and scalar operations between array objects and, occasionally, on reduction operations. Optimizing preprocessors for FORTRAN such as KAPF [16] and VAST-2 [19] do attempt to detect matrix products in loop nests in order to convert them to BLAS operations. However, these preprocessors appear to perform a relatively simple pattern match to accomplish this and do not provide a general solution. In the motivating examples presented in the following section, KAPF was unable to detect any matrix products and VAST-2 only was only successful on the first example. Finally, an analogous problem occurs in the context of automatic parallelization for message passing architectures. Detecting the potential use of high-level primitives such as broadcasts or reductions as opposed to low-level sends and receives can result in substantial savings in the cost of communication [11].

There are four different MATLAB compilers we are aware of: Falcon [5, 6, 7], a research compiler developed at Illinois, compiles MATLAB into FORTRAN 90; MCC [17], from the MathWorks, [2] compiles into C; MATCOM [14], from the Israel Institute of Technology, compiles into C++; and MATCH [22], from Northwestern, compiles directly to special purpose hardware. To generate efficient code, these compilers, to varying degrees, focus upon static techniques to determine information such as shape and type of variables at compile-time. Of these compilers, only Falcon appears to consider high-level optimizations at the MATLAB level. Falcon [5, 7] utilizes an extensible database of pattern matching transformations to be performed interactively by the user on MATLAB code. Pattern replacement allows for a number of optimizations beyond the scope of this paper. However, its syntactic nature limits its ability to optimize loops in the manner we seek. [3]

Similar tools have been developed for symbolic computer algebra systems such as REDUCE and Maple. For example, Gentran [8] generates FORTRAN programs to evaluate REDUCE symbolic expressions much more efficiently than the REDUCE environment itself. Optimizations such as common subexpression elimination are highly critical, and aggressive algorithms for performing them have been developed in this context. Transfor [10], a similar package, maps matrix operations in Maple directly to BLAS operations rather than FORTRAN loops to obtain better performance.

Finally, we cite the classic dynamic programming algorithm for optimally associating a sequence of matrix multiplications [13]. However, this algorithm is limited in its usefulness to compiler optimization by the fact requires exact size information and an explicit matrix formulation as opposed to loop nests.

[2] The MathWorks, Inc. is the developer and producer of MATLAB

[3] We are working with David Padua and his group at Illinois in order to incorporate our work into the next generation Falcon compiler.

```
do i = 1,n
  do k = 1,p
    X(i,k) = B(i,k)
    do j = 1,i-1
      X(i,k) = X(i,k) - L(i,j)*X(j,k)
    enddo
    X(i,k) = X(i,k)/L(i,i)
  enddo
enddo
```

Figure 4: Lower Triangular Solve with multiple right hand sides

3 Motivating Examples

To motivate our framework and transformation algorithms, we present four examples.

3.1 Example 1

Figure 4, a version of lower triangular solve with multiple right hand sides, is a simple example that demonstrates some of the subtleties of detecting matrix products. In this code, we are solving the system $L * X = B$ for X where L is an $n \times n$ lower triangular matrix and B is a dense $n \times p$ matrix. One way of optimizing this code is to realize that the innermost statement is performing a matrix-vector product. Conventional loop transformation technology can isolate the innermost statement as shown in Figure 5. It can be seen that this loop nest is computing a matrix product of the form $X(i, 1:p) = X(i, 1:p) - L(i, 1:i-1) * X(1:i-1, 1:p)$. Automatically detecting this would enable us to convert it to a single BLAS (DGEMV) call[4].

```
do k = 1,p
   do j = 1,i-1
      X(i,k) = X(i,k) - L(i,j)*X(j,k)
   enddo
enddo
```

Figure 5: Example 1

3.2 Example 2

Our next example is taken from the Falcon project's benchmark suite of MATLAB programs [6]. Figure 7 shows the performance bottleneck of a Galerkin method numerical approximation code. Although this loop nest represents only six lines of code out of about fifty in the entire program, it consumes about 97% of the total execution time. The bottom portion of Figure 7 shows an equivalent computation formulated as two matrix products. The result is roughly a 300-fold improvement in the performance of this loop nest and a 30-fold improvement in the entire program!

[4] It should be noted that the original code in Figure 4 actually has a BLAS equivalent (DTRSM). Detecting this is, however, beyond the scope of this paper.

Without loss of generality, we will use the following simplified form of this code as our example:

```
for i=1:N
  for j=1:N
    phi(k) = phi(k) + a(i,j)*xtemp_se(i)*f(j);
  end
end
```

Figure 6: Example 2

As earlier, this code can be obtained by performing conventional vectorization techniques: scalar expansion [25] on `xtemp` and loop distribution on the i loop. In addition, we denote the `cos` expression with the point-wise scalar function `f` for conciseness[5]. Note that this loop nest is not a matrix-vector product; rather it is a vector-matrix-vector product. We must detect this automatically. [6]

3.3 Example 3

Our next example is also taken from the Falcon benchmark suite. Figure 8 highlights a single statement that requires the bulk of the execution time in the Quasi-Minimal Residual benchmark. In this statement, `w_tld`, `q`, and `w` are column vectors; `A` is a two dimensional matrix; and `beta` is a scalar. In the original statement, the computation of `A'*q` requires most of the computation. To compute this expression, the MATLAB interpreter first computes the transpose of the two dimensional matrix `A`, requiring a two dimensional temporary matrix and corresponding work. In contrast, when this expression is converted to the equivalent expression (q'*A)', only one dimensional vectors need to be transposed. The result is roughly a 20-fold improvement in the performance of this statement and a 7-fold improvement in the entire program!

Note that unlike the previous two examples, this example already contains high-level matrix operations. Instead of detecting high-level operations, we wish to utilize the semantics of matrix transpose and product operations.

3.4 Example 4

Finally, we present a more complex example in which detection of matrix products and utilization of corresponding semantic information can achieve an asymptotic improvement in performance. Consider the loop nest in Figure 10, where A, B, and C are $n \times n$ matrices, and y and x are n vectors. Most FORTRAN compilers will optimize address calculations and perform loop transformations like tiling and unrolling, but the amount of computation is still $O(n^4)$. Closer examination reveals that the above code is computing $y = y + A * B^T * C * x$. Computing these high-level matrix products naively in a left to right order results in $O(n^3)$ work, resulting in a gain in asymptotic efficiency. Better still, exploiting matrix product associativity to compute the products in right to left order results in $O(n^2)$ work!

[5] In MATLAB, a scalar function such as *cos* may be applied point-wise over an array.

[6] Note that we are assuming that we have access to shape information. This information is not explicitly available in MATLAB programs, but it is inferred by the FALCON compiler [6]. In this case, it can prove that *phi* is a column vector and that *k*, *y*, and *L* are scalars. *pi* is a scalar constant, and *cos* is a point-wise scalar function.

Original MATLAB Code:

```
                39:   for i=1:N
 0.24s,   1%    40:     xtemp = cos((i-1)*pi*x/L);
 0.23s,   1%    41:     for j=1:N
16.36s,  88%    42:       phi(k) = phi(k) + a(i,j)*xtemp
                                                 *cos((j-1)*pi*y/L);
 1.26s,   7%    43:     end
 0.03s,   0%    44:   end
```

Transformed MATLAB Code:

```
0.01s,   1%   39:   xtemp_se = cos((0:N-1)*pi*x/L);
0.06s,   9%   40:   phi(k) = phi(k) + xtemp_se*a
                                    *cos((0:N-1)*pi*y/L)';
```

Figure 7: Execution Bottleneck of Galerkin Method Code in MATLAB

Original MATLAB Code:

```
19.10s, 70%   50:   w_tld = ( A'*q ) - ( beta*w );
```

Transformed MATLAB Code:

```
0.80s,  9%   50:   w_tld = ( q'*A )' - ( beta*w );
```

Figure 8: Execution Bottleneck of Quasi-Minimal Residual Code in MATLAB

4 The Framework

In this section we define our algebraic language (which we term Abstract Matrix Form, or AMF) and a set of equational axioms for this language that encode semantic information. AMF is a restricted language that provides us with the means of representing loop nests, high-level matrix operations, and a mixture of the two. In our framework, a compiler will translate MATLAB or FORTRAN loop nests or expressions into AMF, perform transformations in AMF, and, finally, translate back into MATLAB or FORTRAN. In this section, we will define the AMF language and relevant axioms. Our axiomatic approach is, by nature, extensible; however we will limit our discussion to those operators and axioms that are useful to the examples in this paper.

4.1 Abstract Matrix Form

AMF expressions are numerical array objects of arbitrary size and number of dimensions (including, of course, scalars values). In contrast to programming language such as FORTRAN or MATLAB, AMF does not specify any ordering between dimensions. That is, there is no notion of a first (or row) dimension, a second (or column) dimension, and so on. AMF expressions can take the following form:

- s : a scalar variable or constant

MATLAB	AMF
`a(i-1,j+1)`	$a_{i-1,j+1}$
`a(1:m,1:n)`	$\forall_{i\in 1:m}\forall_{j\in 1:n}a_{i,j}$
`a(1:m,1:n) + 5`	$(\forall_{i\in 1:m}\forall_{j\in 1:n}a_{i,j}) + (\forall_{i\in 1:m}\forall_{j\in 1:n}5)$
`sum(a(1:m,1:n),1)`	$\sum_i \forall_{i\in 1:m}\forall_{j\in 1:n}a_{i,j}$
`a(1:m,1:n)'*b(1:l,1:m)'`	$P_k(\forall_{k\in 1:m}\forall_{i\in 1:n}a_{k,i}, \forall_{j\in 1:l}\forall_{k\in 1:m}b_{j,k})$

`for i = 1:n`
`  x(i) = x(i) + y(i);`
`end`
	$\forall_{i\in 1:n}x_i = \forall_{i\in 1:n}(x_i + y_i)$

`for i = 1:n`
`  k = k + x(i);`
`end`
	$k = k + \sum_i \forall_{i\in 1:n}x_i$

Table 1: Equivalent MATLAB and AMF computations

```
w_tld = ( A'*q ) - ( beta*w );
```

Figure 9: Example 3

```
do i = 1,n
  do j = 1,n
    do k = 1,n
      do l = 1,n
        y(i) = y(i) + x(j)*A(i,k)*B(l,k)*C(l,j)
      enddo
    enddo
  enddo
enddo
```

Figure 10: Example 4

- a_{ij} : a scalar value denoted by an array variable with scalar indices
- $f(e_1, e_2, \ldots, e_n)$: the point-wise application of the scalar operation f on AMF expressions $e_1, e_2, \ldots, e_n$

  $e_1 \ldots e_n$ and the resulting expression must conform; i.e, they have the same size in each dimension. We will use $+$ to specify point-wise addition and $\cdot$ to specify point-wise multiplication.
- $\forall_{i\in l:u}e$: the expansion of AMF expression e in the dimension corresponding to the scalar integer variable i

  The resulting expression must conform with e with the exception that it has one additional i dimension where i takes integer values between the scalar bounds l and u. Each slice of the resulting expression, for a fixed i, is simply e where the value of i substituted appropriately. In the remainder of this paper, we will omit the bounds of a $\forall$ when it is not relevant to the discussion.
- $\sum_i e$: the additive reduction of the AMF expression e in the i dimension

  The resulting expression conforms e with the exception that all values along the i dimension are collapsed via addition into a single value.
- $P_i(e_1, e_2)$: the product of AMF expressions e_1 and e_2

  e_1 and e_2 may only share i as an expanded dimension and must be the same size along the i dimension. The resulting expression is expanded along the remaining disjoint dimensions of e_1 and e_2. Note that this is a generalization of matrix product.

AMF statements take the form $l = e$, where l is a scalar or array variable expanded by zero or more $\forall$ operations and e is an arbitrary AMF expression. Note that l and e must have identical shape. The semantics of an AMF statement are defined so that e is entirely computed before any memory location specified by l are written.

Table 1 displays examples of MATLAB statements and expressions and the corresponding AMF statements and expressions. We will describe the conversion process between MATLAB and AMF in the next section.

4.2 Definitions

To facilitate our discussion, we define the following terms:

Definition 1 *$dims(e,i)$ is a predicate that is true if and only if e is expanded in the i dimension. $dims(e)$ is the set of all dimensions i, such that $dims(e,i)$ holds.*

Definition 2 *$constant(e,i)$ is a predicate that is true only if e is invariant along the i dimension.*

Definition 3 *$compress(e,i)$, if e is invariant along the i dimension, is the value of e for any fixed i. That is, if $constant(e,i)$ then $compress(e,i) = \hat{e}$ where $e = \forall_i\hat{e}$. Otherwise, if not $constant(e,i)$, define $compress(e,i) = e$. Let $compress(e, \{i_1, \ldots, i_n\}) = compress(compress(\ldots compress(e, i_1)\ldots, i_{n-1}), i_n)$ and $compress(e) = compress(e, dims(e))$.*

Definition 4 *$size(e)$ is the total number of dimensions in which e is expanded.*

Definition 5 *$basesize(e)$ is the number of dimensions, i, in which $constant(e,i)$ is false. In other words, $basesize(e) = size(compress(e))$.*

4.3 Axioms

In order to reason about AMF expressions, we must specify semantic information regarding AMF operations. We represent this information as a set of axioms. Note that this list does not capture all the semantic information available, but only that which we will make use of in this paper.

Axiom 1 $\forall_i f(e_1, e_2, \ldots, e_n) = f(\forall_i e_1, \forall_i e_2, \ldots, \forall_i e_n)$

The result of an expansion of a point-wise operation on n expressions is equivalent to the result of a point-wise operation on n expanded expressions.

Axiom 2 $\forall_i \sum_j e = \sum_j \forall_i e$ *where* $i \neq j$

Provided that they operate over separate dimensions, an expansion and a reduction operator may be interchanged.

Axiom 3 $\forall_{i \in l_1:u_1} \forall_{j \in l_2:u_2} e = \forall_{j \in l_2:u_2} \forall_{i \in l_1:u_1} e$ *where both* l_2 *and* u_2 *are constant with respect to* l_1 *and* u_1 *and vice versa.*

Provided that the bounds of j are not dependent on i or vice versa, two expansion operators may be freely interchanged. The proof is trivial.

Axiom 4 $\sum_i \sum_j e = \sum_j \sum_i e$

Reductions over different dimensions are independent and may be freely interchanged.

Axiom 5 $constant(e_1, i) \Rightarrow$
$\sum_i (e_1 \cdot e_2) = compress(e_1, i) \cdot \sum_i e_2$

Constant expanded expressions may be hoisted outside of a reduction along the same dimension. This follows directly from the distributive law of arithmetic.

Axiom 6 $\sum_j (e_1 \cdot e_2) = P_j(compress(e_1),$
$compress(e_2))$ *if* $dims(compress(e_1)) \cap$
$dims(compress(e_2)) = j$.

This axiom follows directly from the definition of AMF products.

In addition, we do not explicitly state but will assume standard scalar algebraic properties; for example, we will make use of the fact that the point-wise application of a commutative scalar operation is itself commutative.

4.4 Power of the Axioms

The axioms above permit us to relate a high-level matrix operation such as matrix product with its underlying scalar computations. As we shall demonstrate in the next section, these axioms will allow us to map loop-based scalar code into high-level matrix products. At first glance, it may appear surprising that we do not also include such high-level axioms as matrix product associativity, since utilizing such a property is one of our stated goals. However, such an axiom would be redundant. In conjunction with scalar algebraic properties, the axioms above are sufficient to express basic high-level properties of matrix products including, as we shall see in the process of our transformations, associativity and transposed commutativity.

In fact, only one additional axiom

Axiom 7 $\sum_i (e_1 + e_2) = (\sum_i e_1) + (\sum_i e_2)$

is required to arrive at the all of the commutative ring axioms of $n \times n$ matrices over addition and matrix multiplication. The above axiom is not relevant to the examples in this paper, but may used to demonstrate to distributivity of matrix product over addition.

5 Axiom-driven Transformation

At this point, we claim that we have defined enough machinery to reason about an interesting set of matrix computations, namely matrix products. Our language, AMF, provides the means of representing computation in terms of loops, in terms of high-level operations, and in a mixture of the two. For any given AMF expression, our axioms essentially define a space of provably equivalent AMF expressions. That is, by applying a series of axioms (in either direction) as rewrite rules upon an AMF expression, we obtain a mathematically equivalent expression.

What remains is to develop a systematic way of using these axioms to transform computations into more desirable forms. In this section, we describe a heuristic process to accomplish this and motivate it through our four examples.

Step 1 *Conversion to AMF*

The first step is to convert MATLAB or FORTRAN statements and expressions into AMF. As we mentioned earlier, AMF is designed to express and operate over a subset of the computation that may appear in a FORTRAN or MATLAB program. We can represent, as a single AMF statement, a FORTRAN or MATLAB assignment statement that is surrounded by zero or more loops. The loops must carry no flow (read after write) or output (write after write) dependences [7] except those due to an additive reduction. The statement must consist of point-wise array operations, additive reductions, and matrix products. Figure 11 highlights the conversion process from MATLAB to AMF. In this figure, we use the phrase $\text{unify}(v_1, v_2, \ldots, v_n)$ where v_1, v_2 etc are variables to mean "make v_1, v_2 etc the same variable". A small detail is that MATLAB permits us to write expressions of the form `a(1:m,1:n) + 5` since `5` is implicitly expanded into an `m × n` array containing 5. We will assume that this expansion is carried out explicitly during the MATLAB to AMF conversion, as shown in Table 1(example 3), but we have not shown pseudocode for this in Figure 11. The conversion process from FORTRAN is similar.

For conciseness, we assume that all MATLAB data objects are scalars or two-dimensional matrices. [8] We differentiate between row and column vectors in MATLAB [9] by indexing the non-expanded dimension with 1.

The following is the result of converting each of our examples into AMF.

[7] Anti (write after read) dependences are preserved by the semantics of AMF statements.

[8] This is true of MATLAB 4.2, to which the Falcon benchmarks conform. MATLAB 5 and later permit multidimensional arrays and other data objects. However, as matrix products are not defined for these types, we do not consider them.

[9] No such distinction in required in FORTRAN 90.

```
MAT_Expr_to_AMF(e) {
  case (e) {
    scalar c → return (c, 1, 1)

    array point a(i_1, i_2) → return (a_{i_1,i_2}, 1, 1)

    array section a(l_1 : u_1, l_2 : u_2) →
      return (∀_{r∈1:u_1−l_1+1} ∀_{c∈1:u_2−l_2+1} a_{r+l_1−1,c+l_2−1}, r, c)

    point-wise function f(e_1, e_2, ..., e_n) →
      let (ẽ_1, r_1, c_1) = MAT_Expr_to_AMF(e_1)
            :
          (ẽ_n, r_n, c_n) = MAT_Expr_to_AMF(e_n)
          r = unify(r_1,...,r_n)
          c = unify(c_1,...,c_n)
        return (f(ẽ_1, ẽ_2, ..., ẽ_n), r, c)

    transposed expression e' →
      let (ẽ, r, c) = MAT_Expr_to_AMF(e)
        return (ẽ, c, r)

    matrix product e_1 * e_2 →
      let (ẽ_1, r_1, c_1) = MAT_Expr_to_AMF(e_1)
          (ẽ_2, r_2, c_2) = MAT_Expr_to_AMF(e_2)
          i = unify(c_1,r_2)
        return (P_i(ẽ_1, ẽ_2), r_1, c_2)

    :

    else → fail
  }
}

MAT_to_AMF(s) {
  case (s) {
    assignment e_1 = e_2 →
      let (ẽ_1, r_1, c_1) = Mat_Expr_to_AMF(e_1)
          (ẽ_2, r_2, c_2) = Mat_Expr_to_AMF(e_2)
          unify(r_1, r_2)
          unify(c_1, c_2)
        return ẽ_1 = ẽ_2

    dependence free loop for i = l : u, e_1 = e_2, end →
      let (ẽ_1 = ẽ_2) = MAT_to_AMF(e_1 = e_2)
        return (∀_{i∈1:u−l+1} ẽ_1 = ∀_{i∈1:u−l+1} ẽ_2)

    reduction loop for i = l : u, e_1 = e_1 + e_2, end →
      let (ẽ_1 = ẽ_1 + ẽ_2) = MAT_to_AMF(e_1 = e_1 + e_2)
        return (ẽ_1 = ẽ_1 + Σ_i ∀_{i∈1:u−l+1} ẽ_2)

    :

    else → fail
  }
}
```

Figure 11: MATLAB to AMF conversion

1. $\forall_k X_{i,k} = \forall_k (X_{i,k} - \sum_j \forall_j (L_{i,j} \cdot X_{j,k}))$
2. $phi_k = phi_k + \sum_i \forall_i \sum_j \forall_j (a_{i,j} \cdot xtemp\_se_{1,i} \cdot f(j))$
3. $\forall_i w\_tld_{i,1} = P_j(\forall_i \forall_j A_{j,i}, \forall_j q_{j,1}) - (\forall_i beta) \cdot (\forall_i w_{i,1})$
4. $\forall_i y_i = \forall_i (y_i + \sum_j \forall_j \sum_k \forall_k \sum_l \forall_l (x_j \cdot A_{i,k} \cdot B_{l,k} \cdot C_{l,j}))$

Step 2 *Conversion of Products to Reductions*

Next, since our axioms encode the properties of matrix products in terms of underlying scalar operations, we must first convert any product into reductions of point-wise scalar operations. The following pseudo-code describes this process. In this pseudo-code, the phrase child expression of e refers to a subexpression nested immediately inside e.

```
Product_to_Reduction(e) {
  foreach child expression c of e
    c = Product_to_Reduction(c)

  if e == P_i(e_1, e_2) // Axiom 6
    e = Σ_i (∀_{dims(e_2)−{i}} e_1 · ∀_{dims(e_1)−{i}} e_2)
}
```

In our examples, only the third already contains a matrix product. The result of converting this is:

3. $\forall_i w\_tld_{i,1} = \sum_j (\forall_i \forall_j A_{j,i} \cdot \forall_i \forall_j q_{j,1}) - (\forall_i beta) \cdot (\forall_i w_{i,1})$

Step 3 *Expansion Operator Distribution*

This next step is crucial to isolating high-level operations in AMF expressions. Essentially, we are extending loop distribution from just loop nests into AMF expressions themselves. For example, left to right application of Axiom 1 may be used to convert a repeated application of a scalar function f into a single point-wise application. The following pseudo-code utilizes Axioms 1 and 2.

```
Distribute(e) {
  if e == ∀_i ê
    ê = Distribute(ê)
    case(ê) {
      point-wise function f(e_1, ..., e_n) →
        e = f(∀_i e_1, ..., ∀_i e_n) // Axiom 1

      Σ_j e_1 → e = Σ_j ∀_i e_1 // Axiom 2
    }
  foreach child expression c of e
    c = Distribute(c)
}
```

The following are the effects of expansion operator distribution on Examples 1, 2, and 4:

1. $\forall_k X_{i,k} = \forall_k X_{i,k} - \sum_j (\forall_k \forall_j L_{i,j} \cdot \forall_k \forall_j X_{j,k})$
2. $phi_k = phi_k + \sum_i \sum_j ((\forall_i \forall_j a_{i,j}) \cdot (\forall_i \forall_j xtemp\_se_{i,1}) \cdot \forall_i f(\forall_j j))$
4. $\forall_i y_i = (\forall_i y_i) + \sum_j \sum_k \sum_l ((\forall_i \forall_j \forall_k \forall_l x_j) \cdot (\forall_i \forall_j \forall_k \forall_l A_{i,k}) \cdot (\forall_i \forall_j \forall_k \forall_l B_{l,k}) \cdot (\forall_i \forall_j \forall_k \forall_l C_{l,j}))$

Step 4 *Invariant Term Hoisting*

At this point, we arrive at a high-level representation of computation. Now, we seek to realize optimizations. In this step, we hoist constant terms, via Axiom 5, outside of additive reductions. In general, we will have conflicting choices as to which term we may hoist. As we will see later, the choice that we make at this step can determine the associativity of matrix products down the line.

In the pseudo-code below, we attempt to hoist the largest array object as far outside as we can. Intuitively, we are eliminating as much redundant computation in as large an object. Mathematically, we are enabling further optimization in the next step.

Hoist_Terms(e) {
  if $e == \sum_{i_1} \ldots \sum_{i_n} (e_1 \cdots e_m)$
    Pick e_k such that
      i), size(compress(e_k, $\{i_1, i_2, \ldots, i_n\}$)) is minimized
        for $k \in 1 \ldots m$ and
      ii), basesize(e_k) is maximized among remaining terms.

    Reorder $i_1, \ldots, i_n$ to $i'_1, \ldots, i'_l, \ldots i'_n$ so that constant(e_k,i'_j)
      for $j \in 1 \ldots l-1$ and not constant(e_k,i'_j) for $j \in l \ldots n$.

    $e = \sum_{i'_1} \ldots \sum_{i'_{l-1}} (compress(e_k, \{i_l, \ldots, i_n\}) \cdot \sum_{i'_l} \ldots$
        $\sum_{i'_n} (e_1 \cdots e_{k-1} \cdot e_{k+1} \cdots e_m))$ // Axioms 4 and 5

  foreach child expression c of e
    c = Hoist_Terms(c)
}

This step is applicable to Examples 2 and 4:

2. For this example, we may arbitrarily choose the *xtemp_se* or $f(j)$ terms. In both cases, we are dealing with redundantly expanded vectors. In the following, we hoist *xtemp_se*.

$$phi_k = phi_k + \sum_i ((\forall_i xtemp\_se_{i,1}) \cdot \sum_j ((\forall_i \forall_j a_{i,j}) \cdot \forall_i f(\forall_j j)))$$

4. With this example, we may first choose between hoisting the A and x terms outside of two reduction operators. We choose A, the larger object: [10]

$$\forall_i y_i = (\forall_i y_i) + \sum_k ((\forall_i \forall_k A_{i,k}) \cdot \sum_j \sum_l ((\forall_i \forall_j \forall_k \forall_l x_j) \cdot (\forall_i \forall_j \forall_k \forall_l B_{l,k}) \cdot (\forall_i \forall_j \forall_k \forall_l C_{l,j})))$$

Again, we are left with a similar choice between B and x, and, for the same reasons, we choose B this time:

$$\forall_i y_i = (\forall_i y_i) + \sum_k ((\forall_i \forall_k A_{i,k}) \cdot \sum_l ((\forall_i \forall_k \forall_l B_{l,k}) \cdot \sum_j ((\forall_i \forall_j \forall_k \forall_l x_j) \cdot (\forall_i \forall_j \forall_k \forall_l C_{l,j}))))$$

Step 5 *Expansion Operator Hoisting*

In this step, as in the previous one, we eliminate redundant computation. In the case, we hoist $\forall$ operators outside of $\sum$ operators. Note that if $constant(e, j)$, evaluating $\forall_j e$ requires only that we evaluate e once and then copy over the j dimension. In the following pseudocode, we hoist expansion operators:

Hoist_Expansion(e) {
  if $e == \sum_{i_1} \ldots \sum_{i_n} (e_1 \cdots e_m)$ and
      $\exists j \in$ dims(e) such that $j \notin$ dims(compress(e))
    $e = \forall_j \sum_{i_1} \ldots \sum_{i_n}$ (compress(e_1,j) $\cdots$
      compress(e_m,j)) // Axioms 2 and 3

  foreach child expression c of e
    c = Hoist_Expansion(c)
}

This step is applicable to Example 4:

4. $\forall_i y_i = (\forall_i y_i) + \sum_k ((\forall_i \forall_k A_{i,k}) \cdot \forall_i \sum_l ((\forall_k \forall_l B_{l,k}) \cdot \forall_k \sum_j ((\forall_j \forall_l x_j) \cdot (\forall_j \forall_l C_{l,j}))))$

At this point, we have achieved our asymptotic gain in performance in this example. While, even in the previous step, computation required $O(n^4)$ work, hoisting the $\forall_i$ and $\forall_k$ operators has reduced the work to $O(n^2)$.

[10] Although, to be precise, x is expanded to the same size, it contains more replication than A, so we consider it smaller.

Step 6 *Product Conversion*

Finally, now that our computations are in terms of optimized additive reductions, we are ready to convert them into matrix products:

Reduction_to_Product(e) {
  foreach child expression c of e
    c = Reduction_to_Product(c)

  if $e == \sum_i (e_1 \cdot e_2)$ and
      dims(compress(e_1)) $\cap$ dims(compress(e_2)) == $\{i\}$
    $e = P_i$(compress(e_1),compress(e_2)) // Axiom 6
}

The results on our examples are:

1. $\forall_k X_{i,k} = \forall_k X_{i,k} - P_j(\forall_j L_{i,j}, \forall_k \forall_j X_{j,k})$
2. $phi_k = phi_k + P_i((\forall_i xtemp\_se_{i,1}), P_j((\forall_i \forall_j a_{i,j}), f(\forall_j j)))$
3. $\forall_i w\_tld_{i,1} = P_j(\forall_i \forall_j A_{j,i}, \forall_j q_{j,1}) - (\forall_i beta) \cdot (\forall w_{i,1})$
4. $\forall_i y_i = (\forall_i y_i) + P_k((\forall_i \forall_k A_{i,k}), P_l((\forall_k \forall_l B_{l,k}), P_j((\forall_j x_j), (\forall_j \forall_l C_{l,j}))))$

Step 7 *Conversion from AMF*

Finally, we are ready to convert our AMF expressions back into FORTRAN or MATLAB. Figure 12 demonstrates the conversion of AMF into MATLAB. FORTRAN 90, which also allows high-level matrix operations, requires a similar but slightly different process.

It should be noted that, in some cases, it will not be possible to transform an AMF array object to a single FORTRAN or MATLAB array expression. Consider, for example, the AMF expression $\forall_i \forall_j x_{i+j}$. In such a case, we can always fall back upon a loop nest to express the computation.

However, in each of our examples, we may convert directly into MATLAB or FORTRAN 90 statements without loop nests. We arrive at the our desired optimized codes:

1. `X(i,1:p) = X(i,1:p) - matmul(L(i,1:i-1), X(1:i-1,1:p))`
2. `phi(k) = phi(k) + xtemp*(A*cos((0:N-1)'*pi *y/L));`
3. `w_tld = (q'*A)' - beta * w;`
4. `y = y + matmul(A,matmul(transpose(B), matmul(C, x)))`

Note that Figure 12 leaves us with two choices when converting products into MATLAB. Because of the cost of transpose operations in MATLAB, we choose the option that minimizes the number of transposes of two dimensional matrices and, then, the total number of transposes. In Example 3, in particular, we choose to transpose vectors twice rather a matrix once.

AMF_to_MAT($e_1 = e_2$) {
  case (e_1) {
    scalar $s \rightarrow$
      return s = AMF_Expr_to_MAT(e_2,1,1)

    array point $a_{i,j} \rightarrow$
      return $a(i,j)$ = AMF_Expr_to_MAT(e_2,1,1)

    $\forall_{i \in l:u} a_{f(i),k}$ where k is independent of $i \rightarrow$
      return $a(f(l:u),k)$ = AMF_Expr_to_MAT(e_2,i,1)

    $\forall_{i \in l:u} a_{k,f(i)}$ where k is independent of $i \rightarrow$
      return $a(k,f(l:u))$ = AMF_Expr_to_MAT(e_2,1,i)

    $\forall_{i_1 \in l_1:u_1} \forall_{i_2 \in l_2:u_2} a_{f(i_1),g(i_2)} \rightarrow$
      return $a(f(l_1:u_1), g(l_2:u_2))$ =
          AMF_Expr_to_MAT(e_2,i_1,i_2)

    else $\rightarrow$ fail
  }
}

AMF_Expr_to_MAT(e,r,c) {
  case(e) {
    scalar $s \rightarrow$ return s // r == c == 1

    array point $a_{i,j} \rightarrow$ return $a(i,j)$

    $\forall_{r \in l:u} a_{f(r),k}$ where k is independent of $r \rightarrow$
      return $a(f(l:u),k)$

    $\forall_{r \in l:u} a_{k,f(r)}$ where k is independent of $r \rightarrow$
      return $a(k,f(l:u))'$

    $\forall_{c \in l:u} a_{f(c),k}$ where k is independent of $r \rightarrow$
      return $a(f(l:u),k)'$

    $\forall_{c \in l:u} a_{k,f(c)}$ where k is independent of $r \rightarrow$
      return $a(k,f(l:u))$

    $\forall_{r \in l_1:u_1} \forall_{c \in l_2:u_2} a_{f(r),g(c)} \rightarrow$
      return $a(f(l_1:u_1), g(l_2:u_2))$

    $\forall_{r \in l_1:u_1} \forall_{c \in l_2:u_2} a_{g(c),f(r)} \rightarrow$
      return $a(g(l_2:u_2), f(l_1:u_1))'$

    point-wise function $f(e_1, e_2, \ldots, e_n) \rightarrow$
      let $\tilde{e}_1$ = AMF_Expr_to_MAT(e_1,r,c)
          ⋮
          $\tilde{e}_n$ = AMF_Expr_to_MAT(e_n,r,c)
        Compress expanded scalars e_k if possible
        return $f(e_1, e_2, \ldots, e_n)$

    product $P_i(\forall_r \forall_i e_1, \forall_c \forall_i e_2)$ or $P_i(\forall_c \forall_i e_2, \forall_r \forall_i e_1) \rightarrow$
      let $\tilde{e}_1$ = AMF_Expr_to_MAT(e_1,r,i) $*$
            AMF_Expr_to_MAT(e_2,i,c)
          $\tilde{e}_2$ = (AMF_Expr_to_MAT(e_2,c,i) $*$
            AMF_Expr_to_MAT(e_1,i,r))'
          $\tilde{e}$ = The least expensive of $\tilde{e}_1$ and $\tilde{e}_2$.

    ⋮

    else $\rightarrow$ fail
  }
}

Figure 12: AMF to MATLAB conversion

6 Conclusions and Ongoing Work

We have presented a framework for exploiting high-level semantics in numerical codes. In this paper, we have demonstrated the usefulness of our framework in the context of matrix products. As our examples have shown, high-level optimization can result in substantial performance gains. We are currently in the process of implementing this work in the next generation Falcon compiler in conjunction with David Padua and his group at Illinois.

By its very nature, our axiomatic approach can be extended by adding new axioms, extending the space of provably legal transformations. Axiom 7, which is not one of the axioms in our core system, is such an example since it permits the distribution of matrix product over matrix addition, such as converting $(A+B)*C$ to $A*C+B*C$. In general, merely adding such axioms is insufficient since we also need to convey a systematic means of utilizing them to transform programs. For example, given three dense matrices A, B, and C, it is true that $(A+B)*C$, which involves one matrix product and one addition, would require less work to perform than the equivalent $A*C+B*C$, which involves two matrix products and one addition. Yet, this does not mean we should always prefer the former to the latter. A very common computation in numerical linear algebra is that of a rank-one update of the form $(I+u*u')*A$, where A a dense square matrix, I is an identity matrix of the same dimensions, and u is a column vector[9] . In this case, distributing the matrix product over the addition and then reassociating the remaining product (to compute $A+u*(u'*A)$) reduces the required work from $O(n^3)$ to $O(n^2)$. [11] Clearly, to apply a distributive axiom usefully, we will need a precise cost model.

A second direction is to study extensions to our language itself. The purpose of the AMF language is to conveniently represent the subset of computation with which we are interested. In this paper, we have limited ourselves to codes that either have no dependences or very limited dependences in the form of additive reductions. While this may be sufficient to deal with operations such as matrix product, it is inadequate for many other high-level array or matrix operations that are frequently present in libraries or part of languages such as MATLAB. As an example, consider a recurrence of the form $x(i) = k_1 * x(i-1) + k_2$ computed by a MATLAB loop. This computation could be performed much more efficiently via MATLAB's cumulative summation operation. [12]

Finally, we believe that compile-time application of semantic information will be useful in performing locality enhancing optimizations. For example, the most effectively blocked versions of QR and LU factorizations, as shown by Golub and Van Loan [9] and in the LAPACK library [4], are quite different from the standard unblocked version. In both cases, higher-level semantic information about matrix operations is used by algorithm developers to produce better blocked codes. To date, compiler technology has been unable to reason at this level. Our work may enable compilers to accomplish this.

[11] Golub and Van Loan [9] often express rank-one updates in the form $(I+u*u')*A$ when formulating a number of algorithms. It is 'understood' that an implementer should distribute and reassociate the expression by hand.

[12] For vectors `x` and `y`, a cumulative summation may be defined as: `x = cumsum(y)` $\iff x(n) = \sum_{i=1}^{n} y(i)$.

Acknowledgments: We would like to thank Nawaaz Ahmed, George Almasi, Luiz De Rose, Nikolay Mateev, David Padua, and the anonymous reviewers for their helpful comments on earlier versions of this paper.

References

[1] R. C. Agarwal and F. G. Gustavson. *Algorithm and Architecture Aspects of Producing ESSL BLAS on POWER2.*

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[3] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(2):491–542, October 1987.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition.* SIAM, Philadelphia, 1995.

[5] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288. Springer-Verlag, August 1995.

[6] L. De Rose and D. Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *10th ACM International Conference on Supercomputing*, May 1996.

[7] K. Gallivan, B. Marsolf, and E. Gallopoulos. On the use of algebraic and structural information in a library prototyping and development environment. In *Proc. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 4, pages 565–570, Berlin, 1997.

[8] B. Gates. Gentran: An automatic code generation facility for REDUCE. *SIGSAM Bulletin*, 19(3):24–42, August 1985.

[9] G. Golub and C. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1996.

[10] C. Gomez and T. Scott. Maple programs for generating efficient FORTRAN code for serial and vectorised machines. *Computer Physics Communications*, 115:548–562, 1998.

[11] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Supercomputing*, December 1995.

[12] M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4), 1999.

[13] T. Hu and M. Shing. Some theorems about matrix multiplication. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, pages 28–35. IEEE Computer Society, 1980.

[14] Y. Keren. MATCOM: A MATLAB to C++ translator and support libraries. Technical report, Israel Institute of Technology, 1995.

[15] D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Proc. 4th International Computer Software and Applications Conference*, pages 709–715, October 1980.

[16] Kuck and Associates, Inc. KAP for IBM Fortran and C. http://www.kai.com/product/ibminf.html.

[17] The MathWorks, Inc. *MATLAB Compiler*, 1995.

[18] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. Technical report, Cornell University, Department of Computer Science, April 1999. (Forthcoming).

[19] Pacific Sierra Research Corporation. VAST-2 for XL Fortran. http://www.psrv.com/vast/vast_xlf.html.

[20] D. Padua and M. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[21] C. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishing, 1988.

[22] The MATCH Project. A MATLAB compilation environment for distributed heterogeneous adaptive computing systems. http://www.ece.nwu.edu/cpdc/Match/Match.html.

[23] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992.

[24] M. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[25] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.