# Algorithm 940: Optimal Accumulator-Based Expression Evaluation through the Use of Expression Templates

BLAKE NELSON and ROBERT M. KIRBY, SCI Institute, University of Utah
STEVEN PARKER, NVIDIA

In this article we present a compile-time algorithm, implemented using C++ template metaprogramming techniques, that minimizes the use of temporary storage when evaluating expressions. We present the basic building blocks of our algorithm—transformations that act locally on nodes of the expression parse tree—and demonstrate that the application of these local transformations generates a (nonunique) expression that requires a minimum number of temporary storage objects to evaluate. We discuss a C++ implementation of our algorithm using expression templates, and give results demonstrating the effectiveness of our approach.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: Mathematical Software—*Efficiency*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Algorithms*; *labeling*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: C++, dense linear algebra operations, expression templates, template metaprogramming, temporary storage minimization

**ACM Reference Format:**
Nelson, B., Kirby, R. M., and Parker, S. 2014. Algorithm 940: Optimal accumulator-based expression evaluation through the use of expression templates. *ACM Trans. Math. Softw.* 40, 3, Article 21 (April 2014), 21 pages.
DOI: http://dx.doi.org/10.1145/2591005



## 1. INTRODUCTION

Domain-specific languages (DSL) are programming languages that provide techniques and constructs tailored for problem solving in specific problem domains. These types of languages allow programmers to focus on solving problems in familiar terms, and shield them from the details of the underlying machine implementation. General-purpose programming languages such as C, Fortran, and C++, on the other hand, require that users understand the underlying machine implementation and write programs in those terms. Programs written in these general-purpose languages must be translated from the problem domain into a format that can be understood by the machine, which increases both the time required to write the code and the possibility of defects.

A common way to create a DSL is to embed it into a general-purpose host language such as C++, resulting in a domain-specific embedded language (DSEL). A DSEL allows users to take advantage of the strengths of both platforms—the domain-specific interface provided by the DSEL, and access to machine-level details provided by the

---

B. Nelson is currently affiliated with the Space Dynamic Laboratory, Utah State University Research Foundation.
Authors' addresses: B. Nelson (corresponding author), Space Dynamics Laboratory, Utah State University Research Foundation, UT; email: bnelson@cs.utah.edu; R. M. Kirby, The School of Computing and Scientific Computing and Imaging Institute, University of Utah, UT; S. Parker, NVIDIA, Santa Clara, CA.

DOI: http://dx.doi.org/10.1145/2591005

host language. Additionally, a well-chosen host language will already have a wide audience, making it easier for the DSEL to gain acceptance and widespread use.

DSELs are often used to implement simulations using finite-element methods [Christophe 2006; Di Pietro and Veneziani 2009; Kirby 2003]. For this problem domain, users want a DSEL that handles linear algebra constructs, provides linear algebra operations, and can solve systems of linear equations. Consider the following expression that can be found in a typical finite-element program which computes the modal coefficients $\hat{u}$ for a function evaluated at a set of quadrature points [Karniadakis and Sherwin 1999]

$$\hat{u} = \left(B^T W B\right)^{-1} B^T W u,$$

where $B$ and $W$ are matrices and $u$ and $\hat{u}$ are vectors. This expression, when implemented in C without the benefit of a DSEL, will generally be implemented using the Basic Linear Algebra Subprograms library (BLAS) [Lawson et al. 1979] (for performance reasons) as follows.

```
double* B, W, u, u_hat;
int s;

// temporary memory to hold intermediate results.
double* w1, w2;
int* pivot, info;

dgemm('T', 'N', s, s, s, 1.0, B, s, W, s, 0.0, w1, s);
dgemm('N', 'N', s, s, s, 1.0, w1, s, B, s, 0.0, w2, s);
dgetrf(s, s, w2, s, pivot, info);
dgetri(s, w2, s, pivot, w1, s, info);
dgemm('N', 'T', s, s, s, 1.0, w2, s, B, s, 0.0, w1, s);
dgemm('N', 'N', s, s, s, 1.0, w1, s, W, s, 0.0, w2, s);
dgemv('N', s, s, 1.0, w2, s, u, 1, 0.0, u_hat, 1);
```

This code performs the desired computation, but involves a significant amount of machine knowledge, such as the structure and layout of the data in memory, which makes it difficult to write, understand, and debug. In contrast, the same expression, when implemented in C++ with a linear algebra DSEL, is much easier to create and understand as it closely mimics the mathematical expression.

```
Matrix B, W;
Vector u, u_hat;

u_hat = (B^T*W*B)^(-1) * B^T*W*u;
```

This code is easier to write, maintain, and verify. However, the way in which C++ expressions are evaluated introduces a performance penalty that dissuades some users from using a DSEL embedded in C++. The root of this penalty is that expressions are evaluated locally, with the results of each local operation combining to produce the desired global result. This causes performance penalties in two ways: first, many expressions cannot be evaluated optimally in a local fashion; and second, each local operation must store the results of the operation in temporary storage, even if it is not necessary from the mathematics of the operation.

A common way to address these problems in a linear algebra context is through a technique called *loop fusion*. Loop fusion works by combining the local loops corresponding to each operation with a global loop corresponding to the entire expression.

This evaluation plan is able to provide performance improvements by creating an optimal plan based on the global structure of the expression and by reducing or eliminating the necessity of temporary objects [Sanderson 2010; Veldhuizen 1995, 1998]. This approach works well for expressions in which each element of the expression's output can be calculated independently of other elements. This approach does not apply, however, in cases where the output from the operation is a result of a computation involving all elements of the input. As an example, the basic operations on sets, such as union and intersection, cannot be evaluated using loop fusion.

In this article we describe a general-purpose expression template framework that is capable of minimizing the performance penalty caused by unnecessary temporaries by minimizing the number of temporaries created during expression evaluation. This is accomplished via manipulation of the expression parse tree and the application of custom evaluators during expression evaluation. Since not all expressions are best evaluated using accumulators, our library is also capable of detecting and applying loop fusion to an expression tree.

The remainder of this work is organized as follows. In Section 2, we present an overview of existing expression template libraries and how they are used in linear algebra applications. In Section 3, we present an overview of how expression templates work, and describe the C++ structures and features that are common to all expression template libraries. In Section 4, we discuss the details and theory behind the parse tree manipulation that we use to optimize expression evaluation. In Section 5, we discuss our implementation of these concepts. Finally, we show performance results of this library when evaluating common linear algebra expressions in Section 6.

## 2. PREVIOUS WORK

Expression templates are a family of techniques that use C++ templates to separate expression specification and execution. This separation is beneficial for two reasons: first, it allows expressions to be evaluated without the construction of potentially expensive temporary objects; and second, it allows for the creation of optimized execution plans for the expression based on its global structure.

Expression templates were first used to optimize expressions that operate on arrays of numbers [Vandevoorde and Josuttis 2002; Veldhuizen 1995, 1998] such as $u + v + w + z$, which, when evaluated using standard C++ operators, is evaluated as follows

```
Vector t1 = u + v;
Vector t2 = t1 + w;
Vector result = t2 + z.
```

Evaluating the expression in this way requires the construction of two temporary variables, t1 and t2. For small arrays, the cost of constructing these temporaries can exceed the cost of performing the computation. For large arrays, these temporaries can hinder overall program execution by using so much memory that memory used in other parts of the program can be swapped to disk. These temporaries can also introduce unnecessary memory traffic as intermediate values are written to main memory then read back for the next stage of computations.

An alternative evaluation strategy is to perform the requested operation inside a single loop over each index of each array, as follows.

```
for(int i = 0; i < u.size(); ++i)
{
    result[i] = u[i] + v[i] + w[i] + z[i];
}
```

In this manner, temporary variables are eliminated and memory traffic is minimized. We refer to this approach as *loop fusion*. This basic idea has been extended to parallel architectures for even better execution performance [Plagne et al. 2009].

Additional libraries have been created that extend expression template functionality to expressions with mixed Matrix and Vector operations [Guennebaud et al. 2010; Kirby 2003; Walter and Koch 2002]. Other libraries [Lehn et al. 2005; Sanderson 2010] handle mixed expressions and also are capable of evaluating applicable portions of the expression using BLAS.

Some operations and data types do not support the loop fusion approach. Examples include chained matrix multiplication, vector cross-product, and set union and intersection. For these types of expressions, temporary reduction can be achieved by recognizing that some temporaries in a mathematical expression are dictated by the manner in which the expression is written, and are not a requirement of the C++ language. To illustrate this point, consider the expression $(a + b) + (c + d)$, where the operands are not numeric arrays and are not eligible for loop fusion. Standard mathematical evaluation rules require that $(a + b)$ and $(c + d)$ are evaluated independently, which requires a temporary. However, if we know the properties of the $+$ operator, it may be possible to reduce the number of temporaries. In this case, $+$ is associative, so the expression can be rewritten as $a + b + c + d$, which does not require temporaries. This type of transformation is accomplished by manipulating the expression's parse tree and was first developed by Sethi and Ullman [1970], who use parse tree manipulation to minimize register usage in code generated by compilers. In this work, we extend the Sethi and Ullman parse tree manipulations to include additional operator properties and apply them to an expression template framework.

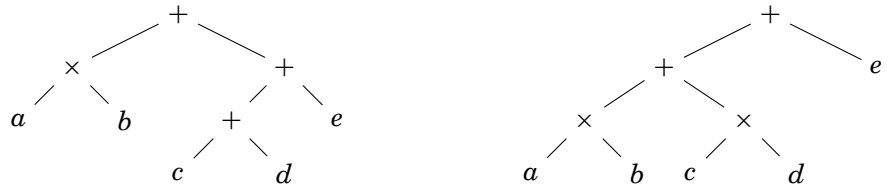
The next version of the C++ standard, C++11, has a feature called *rvalue references* that can be used to define *move constructors*. Move constructors are called when an object is being created as a copy of a temporary. Since the variable that is being copied is a temporary, the new object can, instead of copying the values from the temporary, take ownership of the temporary's internals. This can in some cases provide significant performance improvements.

## 3. EXPRESSION TEMPLATE FUNDAMENTALS

The purpose of all expression template libraries is to improve the performance of expression evaluation in a manner transparent to the user (i.e., the user writes expressions in the same manner as they would without the library). They do this by dividing expression evaluation into two steps, namely, expression creation and expression evaluation. This differs from how expressions are evaluated using the C++ runtime, where the creation and evaluation of an expression is interleaved. The advantage provided by separating these two steps it that it allows the programmer to obtain insight into the expression's structure, which leads to the ability to create accelerated execution strategies for the expression.

### 3.1. Expression Creation

The first phase is the creation of an explicit representation of the expression's parse tree, where literals are located at each leaf and operators are located at each internal vertex (Figure 1). The value at a node is calculated by evaluating its children, then applying its operator to the resulting values. This procedure is, in fact, the way in which the C++ runtime evaluates a node, by first evaluating a node's children, storing the resulting values in temporary objects, then applying the operator to these values. The disadvantage to this approach is that the programmer is not provided with any access to the structure of the parse tree. Therefore, when the operator is executed, there is no information about which node in the tree is being evaluated, nor is there




(a) parse tree for the expression $ab + ((c + d) + e)$

(b) parse tree for the expression $ab + cd + e$

Fig. 1. Two expressions and their parse trees.

any information about the children, such as whether or not they are literal values from the expression or are temporary objects created during previous evaluations.

Expression template libraries are characterized by encoding the structure of the parse tree as template parameters rather than constructing a runtime parse tree. Representing the parse tree as template parameters allows the library to perform many optimizations on the tree at compile time, providing performance improvements over straight expression evaluation. All libraries described in Section 2 and the library described in this work represent parse trees in this manner.

Parse trees are created by modifying C++ operator definitions so that, instead of calculating and returning the operation's result, they create and return the associated node in the parse tree. For example, an addition operator for Vector objects that supports expression templates is written as follows.

```
Node<Node<Vector>, AddOp, Node<Vector> >
operator+(const Vector& lhs, const Vector& rhs)
{
    return Node<Node<Vector>, AddOp, Node<Vector> >(lhs, rhs);
}
```

The operator's return value is a Node object which encodes the structure of the parse tree in the template parameters. The template parameters <Node<Vector>, AddOp, Node<Vector> > indicate that the node is a binary node, with Vector children that will be added together.

To create nodes in a parse tree that have arbitrary parse trees as children, expression template libraries provide a complete set of operators for arbitrary node combinations, such as the following operator which joins two parse trees with an addition node.

```
template<typename L, typename R>
Node<Node<L>, AddOp, Node<R> >
operator+(const Node<L>& lhs, const Node<R>& rhs)
{
    return Node<Node<L>, AddOp, Node<R> >(lhs, rhs);
}
```

After making these changes, an expression no longer produces the results of the expression, but instead produces the parse tree representing the expression. This allows us to modify the tree for optimal evaluation, as we will discuss in Section 4.

### 3.2. Expression Evaluation

The second phase consists of evaluating the expression using the parse tree obtained in the first phase to optimize evaluation. Parse trees are useful because they transform

the task of evaluating an expression into one of tree traversal, which we will discuss in more detail in Section 4.

The most common way to convert this parse tree into a value is to add a copy constructor and assignment operator to the relevant classes that take parse trees as parameters.

```
class Vector
{
    public:
        template<typename T>
        Vector(const Node<T>& rhs)
        {
            rhs.Evaluate(*this);
        }
};
```

In this manner, the user of the expression template library does not interact with the parse tree, and will often not know that an expression template library is in use. Expression evaluation is handled differently in each library and is the primary way in which the libraries differentiate themselves from each other. The advantage of separating expression representation and evaluation is that it allows for the application of performance optimizations based upon the overall structure of the parse tree. In contrast, expressions processed and evaluated without expression templates (such as expressions evaluated directly by the C++ compiler) are unable to perform these optimizations, which can result in significant performance degradation.

## 4. PARSE TREE MANIPULATION AND EVALUATION

Our expression template library accelerates expression evaluation in two ways. First, it manipulates the expression's parse tree to minimize the number of temporaries required by an accumulator-based evaluation strategy. Second, it optimizes runtime evaluation of the parse tree when the tree corresponds to known optimal evaluation routines.

### 4.1. Parse Tree Manipulation

Every expression corresponds to a single parse tree. However, if the expression contains commutative or associative operators, it is possible to rewrite the expression. This rewritten expression will correspond to a different parse tree that evaluates the same overall expression result, but does so differently and, in some cases, requires a different number of temporary variables. In this work, we support tree transformations associated with the associative and commutative operators since they are the transformations most frequently encountered in expressions.

Before discussing the details of the parse tree manipulations we support, we first discuss the evaluation strategy that we will be using. Many existing expression template packages use the loop fusion strategy mentioned earlier. This approach has shown performance gains for array- and vector-based mathematics, but is unable to extend to arbitrary data types (in particular, expressions using a mixture of matrix and vector operations). To address this shortcoming, we use a hybrid evaluation approach based upon accumulator evaluation. In our approach, each parse tree is evaluated using an accumulator that traverses the parse tree. Details are given in Algorithm 1.

What we can see from this algorithm is that a temporary object is required only when the right node is not a leaf. This observation leads to the tree manipulation approach developed in this section, whose goal is to produce a tree with the fewest number of non-leaf nodes on the right side. Also note how this differs from how expressions are

**ALGORITHM 1:** Accumulator-Based Evaluation Stragegy (AEVAL)

**Input**: A parse tree $P$ representing an expression, with root node $\eta$ and accumulator $A$.
**Output**: Expression result stored in $A$.

**if** $\eta$ *is a leaf node* **then**
  $A = \eta$.
**else if** $\eta$ *is a unary node with child* $\eta_0$ *and operator* $\ominus$ **then**
  AEVAL($\eta_0$, $A$)
  $A = \ominus A$
**else if** $\eta$ *is a binary node with left child* $\eta_0$, *right leaf child* $\eta_1$, *and operator* $\oplus$ **then**
  AEVAL($\eta_0$, $A$)
  $A \oplus= \eta_1$.
**else if** $\eta$ *is a binary node with left child* $\eta_0$, *right non-leaf child* $\eta_1$, *and operator* $\oplus$ **then**
  AEVAL($\eta_0$, $A$)
  $T$ =CreateTemporaryAccumulator($\eta_1$)
  AEVAL($\eta_1$, $T$)
  $A \oplus= T$.
**end**

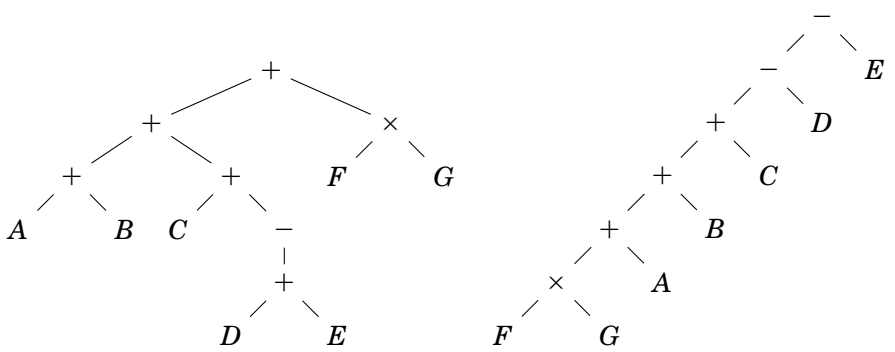



Fig. 2. (Left) The parse tree associated with the expression $((A+B)+(C+-(D+E)))+FG$ that requires two temporaries. (Right) A restructured parse tree for this expression that requires no temporaries.

evaluated in the C++ runtime, where a temporary is created before traversing the left branch as well. It is true that this accumulator-based approach will not perform as well as the loop fusion approach described previously when dealing with array and vector expressions, so our evaluation approach has the ability to override the accumulator approach with more efficient approaches as needed. In particular, we support loop fusion and BLAS overrides.

We now discuss the use of parse tree transformations to reduce the number of temporaries required by an expression. As a motivating example, consider the expression $R = ((A+B)+(C+-(D+E)))+FG$, the parse tree of which is illustrated in Figure 2. When evaluated according to Algorithm 1, this parse tree requires two temporaries. However, an equivalent expression can be written as $R = (FG)+(A+B+C-D-E)$, which does not require any temporaries.

This example illustrates that minimizing the number of temporaries required to evaluate an expression requires parse tree manipulations as well as accumulator-based tree traversal. The remainder of this section describes the available transformations and how they can be used to generate parse trees that require the minimal number of temporaries. These transformations are extensions of the parse tree manipulations developed by Sethi and Ullman [1970]. Their algorithm minimizes the number


Fig. 3. The parse tree for the expression $A \oplus B$ before (left) and after (right) a commutative transform.

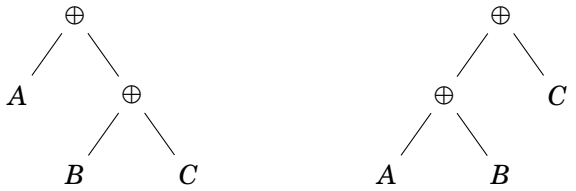



Fig. 4. The parse tree for the expression $A \oplus (B \oplus C)$ before (left) and after (right) an associative transform.

of registers required to evaluate arithmetic expressions composed of operators that are commutative, commutative, and associative, or neither commutative nor associative, while our library supports other combinations, such as operators that are only associative and expressions with a mixture of data types.

### 4.2. Parse Tree Transformations

In this section, we show how, starting with an initial parse tree for an expression, we can apply a finite number of parse tree transformations to produce another tree that allows us to evaluate the expression with a minimal number of temporaries.

A family of parse trees is the set of all parse trees that evaluate an expression

$$F = \{P \in T | P \text{ evaluates } E\},$$

where $E$ is the expression, $P$ is a parse tree, $T$ is the set of all parse trees, and $F$ is the family. The parse trees within a family that require the fewest number of temporaries is defined by

$$F_m = \{P \in F | \forall R \in F : L(P) \leq L(R)\},$$

where $L$ is the count of the number of temporaries required by the parse tree.

A parse tree in a family can be converted to any other parse tree in the family through a finite number of applications of the commutative and associative transformations that are described shortly. Therefore, for any expression, it is possible to generate a parse tree that requires the fewest temporaries by converting the tree into one that is in $F_m$.

An operation is commutative if the order of the operands can be interchanged

$$A \oplus B = B \oplus A. \tag{1}$$

The commutative operator can be represented by a commutative transform in the parse tree, which swaps a binary node's children (Figure 3).

Associative operators are defined by the following property.

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C \tag{2}$$

The associative transform corresponding to an associative operator is shown in Figure 4.

It is convenient to group connected nodes of the same operators into groups called clusters. A cluster is a set of nodes in the tree that have the following properties: each node in the cluster is associated with the same operator, and each node in the cluster is connected and forms a tree.

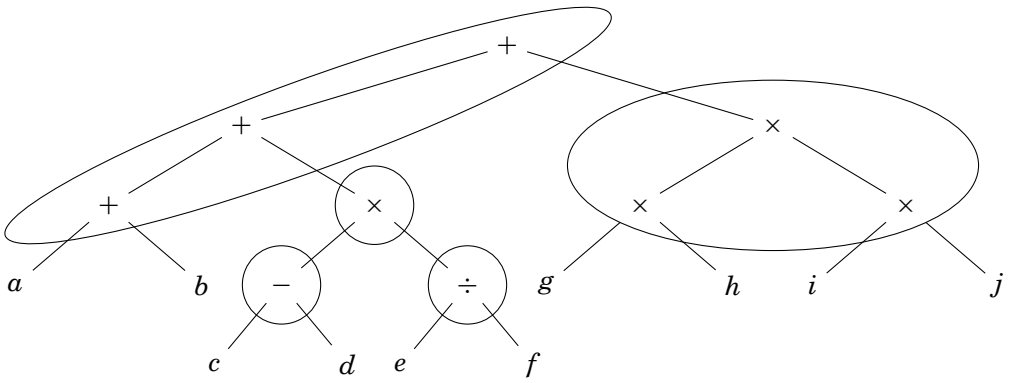



Fig. 5. The clusters for the expression $(a + b) + (c - d)(e/f) + (gh)(ij)$.

**ALGORITHM 2:** Temporary Minimization for Commutative Clusters

**Input**: A commutative cluster
**Output**: Modified cluster requiring the minimum number of temporaries

**foreach** *node in the cluster* **do**
  **if** *Left child is a leaf and right child is not a leaf* **then**
    Apply commutative transformation.
  **end**
**end**

A tree can contain several clusters, as shown in Figure 5. The root of the cluster is the node that precedes all other nodes. The descendants of the cluster are the descendants of the cluster's leaves. A maximal cluster is one that is not a proper subset of any other cluster. A cluster is a commutative cluster if its operator is commutative, an associative cluster if its operator is associative, and an associative-commutative (AC) cluster if its operator is both associative and commutative.

We next provide algorithms for minimizing the number of temporaries required by each type of cluster.

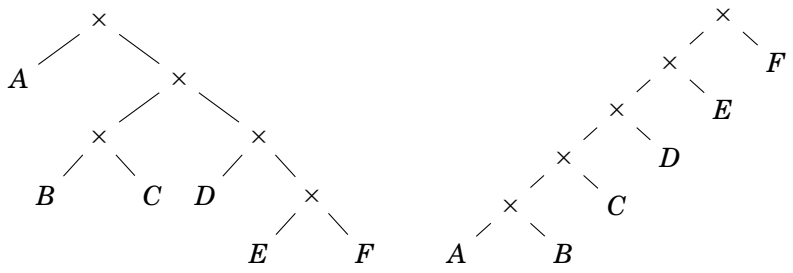
*4.2.1. Commutative Clusters.* An algorithm for minimizing the number of temporaries required by a commutative cluster is given in Algorithm 2.

LEMMA 4.1. *Algorithm 2 minimizes temporaries for commutative clusters.*

PROOF. Each node in a commutative cluster falls into one of three categories.

(1) Both children are leaves. In this case, the node introduces 0 temporaries.
(2) Both children are internal nodes. In this case, the node introduces 1 temporary. Applying a commutative transform to this node does not change the number of temporaries.
(3) One child is a leaf and one is an internal node. In this case, the node can produce either 0 temporaries or 1 temporary, depending on the location of the leaf. Applying a commutative transform to this node when the left child is a leaf removes the temporary and produces a node requiring 0 temporaries.

The only scenarios where temporaries can be reduced is contained in case 3. Our algorithm traverses the tree and identifies all nodes that belong to this case and performs the commutative transform as needed. □




(a) the initial parse tree for the expression $A\,(BC(D(EF)))$

(b) the canonical cluster

Fig. 6. Example of an associative cluster and its corresponding canonical associative cluster.

**ALGORITHM 3:** Generation of Canonical Associative Clusters

**Input**: An associative cluster
**Output**: Modified associative cluster in canonical form

Starting from the cluster's root node:
**if** *Node has 2 leaf children* **then**
  Return
**else if** *Node has a leaf right child and arbitrary left child* **then**
  Apply this algorithm to the left child.
**else if** *If both children are arbitrary trees* **then**
  Apply this algorithm to the right child.
  Apply the associative transform to this node.
  Apply this algorithm to the left child.
**end**

*4.2.2. Associative Clusters.* In order to minimize temporaries in associative clusters, it is useful to first transform the cluster into a form where each operator (except the root) is the left child of an operator, and its right child is one of the cluster's descendants. We call this tree the canonical associative cluster (Figure 6). Canonical associative clusters are generated using Algorithm 3.

LEMMA 4.2. *Algorithm 3 produces the canonical associative cluster for any input associative cluster.*

PROOF. We shall prove this by induction.

*Base Case.* A tree of depth 1 is an associative cluster with one operator and two leaf nodes. This cluster is already in canonical form.

*Induction Step.* Assume this algorithm produces canonical clusters for trees of depth $n-1$. Given a tree of depth $n$:

(1) *Case 1 - Right child is a leaf*. We apply this algorithm to the left child of depth $n-1$.
(2) *Case 2 - The right and left children are arbitrary trees*. We first apply this algorithm to the right child of depth $n-1$, producing a canonical tree. By applying the associative transform to the current node, the new right child is either a leaf or an internal node representing a different operator. In either case, after applying this algorithm to the left child, the result is a canonical tree. □

LEMMA 4.3. *The minimum number of temporaries required by an associative cluster is the number of temporaries required by its canonical cluster and is equal to the number of non-leaf descendants if the leftmost descendant is a leaf, and one less than the number of non-leaf descendants if the leftmost descendant is not a leaf.*

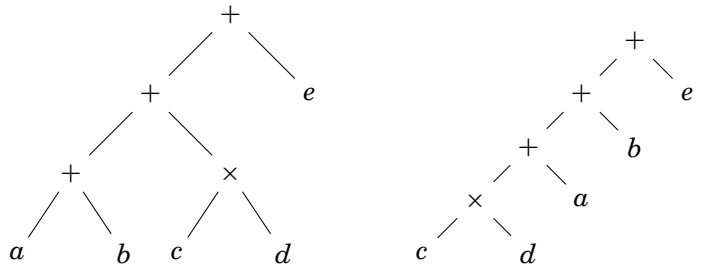



Fig. 7. (Left) Associative-commutative cluster after applying Algorithm 3. This tree contains one temporary, but through repeated used of the commutative property, the node representing *cd* can be moved to the far left, removing the temporary (Right).

**ALGORITHM 4:** Generation of Canonical AC Clusters

**Input**: An AC cluster
**Output**: Modified AC cluster in canonical form

Starting from the cluster's root node:
(1) Generate the canonical associative cluster using Algorithm 3.
(2) Locate the descendant that requires the most temporaries, and swap it and the leftmost descendant.

PROOF. An associative cluster consists of $n$ associative nodes. Of these nodes, $n-1$ of the associative nodes are also interior nodes (one is the root of the tree). This cluster has $n+1$ descendants, with $n_i$ interior nodes and $n_l$ leaf nodes.

Each of the $n$ associative nodes in the cluster has 2 children, for a total of $n$ left children and $n$ right children. Temporaries are minimized when as many interior nodes as possible are left children. The total number of interior nodes is $n_i + n - 1$. Since there are $n$ possible left children, the minimum number of temporaries is $n_i + n - 1 - n = n_i - 1$. This is only achievable in the canonical associative tree. □

*4.2.3. Associative-Commutative Clusters.* The difference between an AC cluster and an associative cluster is that the ordering of the descendants can be modified in an AC tree through repeated use of the commutative transform. In Figure 7, we show an AC cluster that has already been transformed to associative canonical form. In this form, evaluation of the tree requires a single temporary. However, since the operator is an AC operator, we can move the *cd* term to the far left, resulting in an expression that requires no temporaries. The algorithm to perform this transformation is given in Algorithm 4.

### 4.3. Temporary Minimization Algorithm

We now present the temporary minimization algorithm for arbitrary parse trees (Algorithm 5).

LEMMA 4.4. *Minimizing the number of temporaries in each maximal cluster using Algorithm* 5 *produces a parse tree that requires the minimum number of temporaries to evaluate the associated expression.*

PROOF. Assume that the parse tree produced by Algorithm 5 does not require the minimum number of temporaries. This implies that there is a node in the tree for which the application of a commutative or associative transform would reduce the number

**ALGORITHM 5:** Temporary Minimization for Expression Parse Trees

**Input**: Parse tree $P$
**Output**: Modified parse tree $P_m$ that requires the minimal number of temporaries

(1) Identify all maximal clusters in the tree.
(2) Optimize each maximal cluster based upon the operator, using the algorithms described in the previous section.

of required temporaries. This is not possible since all temporaries that can be remove through the use of commutative or associative transforms have been removed. □

## 5. IMPLEMENTATION IN C++

In this section, we present the key components of our expression template library. We note that the final expression template library is too large to justify an exhaustive explanation here, so we focus on the key concepts we use and refer the reader to the source code for further details. Our library was implemented as a part of the Nektar++ spectral/*hp* finite-element framework that can be found at http://www.nektar.info. It exists in its own source directory and can be copied into other projects without the need for Nektar++.

Expression evaluation consists of a compile-time component and a runtime component. The compile-time component performs the following tasks:

— creates the initial, unoptimized parse tree for the expression (Section 5.1);
— manipulates the parse tree to create the optimized tree (Section 5.2);
— assigns an optimal evaluator to each node (Section 5.3).

The runtime component performs the following tasks:

— assigns operands to terminal nodes in the tree;
— traverses the tree in a depth-first manner, evaluating each node according to the optimal evaluator assigned at compile time.

Each of these steps is described in further detail next.

### 5.1. Parse Trees

Parse trees are represented by the template class Node.

```
template<typename ChildNode0, typename Op, typename ChildNode1>
struct Node;
```

Similar to existing expression template libraries, the structure of the tree is encoded in the class' template parameters. Our library supports leaf, unary, and binary nodes, which are specified as indicated in the following table.

| Node Type | C++ Representation |
|---|---|
| Terminal | Node<Matrix, **void**, **void**> |
| Unary | Node<Node<Matrix, **void**, **void**>, NegateOp, **void**> |
| Binary | Node<Node<Matrix, **void**, **void**>, AddOp, Node<Matrix, **void**, **void**> > |

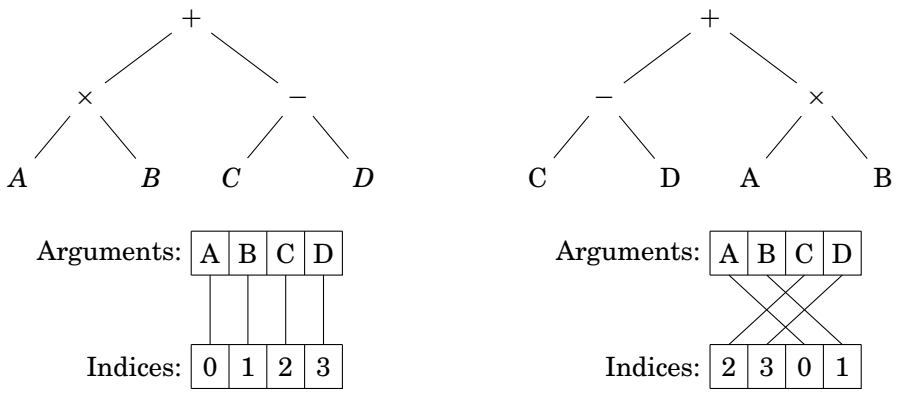



Fig. 8. (Left) Original tree (Right) Tree after commutative transform. The transform modifies the structure of the parse tree and the argument index list, but does not modify the list of arguments.

In addition to the compile-time information regarding parse tree structure, each node must also store the expression's operands. Since the operands are not known at compile time, they are stored in a runtime data structure attached to the node. We use the `boost::fusion` library) to store the heterogeneous list of operands

```
template<typename T1, typename Op, typename T2>
struct Node
{
    typedef typename boost::fusion<...> VectorType;
    typedef typename boost::mpl::vector_c<int, n> IndicesType;
};
```

where the template arguments to the `boost::fusion` object represent the data types of the expression's arguments.

The IndicesType member referenced before maintains a compile-time index list of where each operand fits in the parse tree's structure. When the node is first created, the ordering of the arguments corresponds to the ordering of the arguments in the original expression (Figure 8-Left). When transformations are made to the tree (as shown in Figure 8-Right, where a commutative transform is performed), we want to avoid modifying the runtime list of arguments, as this could negate the performance improvements we are trying to achieve. Instead, we modify the compile-time index list to point to the correct nodes in the argument list. In this way, we can make arbitrary changes to the parse tree without needing to perform any work at runtime to modify the argument order.

### 5.2. Manipulating the Parse Tree

Manipulating the parse tree is done through the use of class templates called *metafunctions*, where the function parameters are expressed as template parameters, and the function results are expressed in terms of nested typedefs. For parse tree manipulation, most metafunctions in our library accept the parse tree to be transformed and the argument index list as parameters, and return the modified parse tree and index list as nested typedefs.

To illustrate, consider the metafunction for performing commutative transforms on a tree.

```
// Base case representing identity transformation.
template<typename NodeType, typename IndicesType, unsigned int
    IndexStart, typename enabled=void>
struct CommutativeTransform
{
    typedef NodeType TransformedNodeType;
    typedef IndicesType TransformedIndicesType;
};

// Case where the operator is commutative.
template<typename Left, typename Op, typename Right, typename
     IndicesType, unsigned int IndexStart>
struct CommutativeTransform<Node<Left, Op, Right>, IndicesType, IndexStart,
    typename boost::enable_if<CommutativeTraits<typename Left::ResultType,
        Op, typename Right::ResultType> >::type>
{
    typedef Node<Right, Op, Left> TransformedNodeType;
    typedef typename Swap<IndicesType>::type TransformedIndicesType;
};
```

In this example, the unspecialized version of the class does not modify the parse tree or the indices, and represents applying the commutative transform to a node in which the commutative property does not hold. The specialized version of the class is only valid when the operator is commutative, and it swaps the nodes in the parse tree and swaps the indices as well.

We can also see from this example that our transformations are only performed when the operator supports the corresponding property. Our library makes extensive use of traits classes to supply this information at compile time. Our traits classes accept operand data types as parameters rather than nodes in the parse tree. In this way, the traits can be used in contexts outside of our expression template library, as well as provide information about the expression independently of the parse tree's structure. In this example, we supply the CommutativeTraits class that indicates which combinations of operators and data are commutative. The default behavior is to mark addition and multiplication as commutative.

```
template<typename L, typename Op, typename R>
struct CommutativeTraits : public boost::false_type {};

template<typename R, typename T>
struct CommutativeTraits<R, AddOp, T> :
    public boost::true_type {};

template<typename R, typename T>
struct CommutativeTraits<R, MultiplyOp, T> :
    public boost::true_type {};
```

To override this behavior, users of the library must create their own specialization for their specific data types. So, for example, since matrix multiplication is not

commutative, we must override this traits class to prevent commutative transforms from being applied to these types of matrix expressions.

```
template<>
struct CommutativeTraits<Matrix, MultiplyOp, Matrix> :
    public boost::false_type {};
```

There is a similar class called AssociativeTraits that follows the same convention.

It is beyond the scope of this article to provide the source code for all of the metafunctions and traits used by our library, but full documentation can be found in the source code.

### 5.3. Evaluation Optimizations

Eliminating temporaries is a useful first step, but there are additional ways in which expression evaluation can be made faster. While the accumulator-based approach helps to reduce the total number of temporaries required, in some cases there are better evaluation methods than the accumulator-based approach. For example, the sum of arrays can be done much more efficiently through a loop fusion approach than an accumulator, and expressions such as $\alpha AB + \beta C$, where $A, B, C$ are matrices, can be evaluated much more efficiently through a call to the BLAS function dgemm rather than accumulator-based approaches.

To allow for data-dependent optimizations, our expression template library relies on template specialization. Our library defines an override class as follows.

```
template<typename LhsType, typename Op, typename RhsType, typename
    enabled=void>
struct BinaryBinaryEvaluateNodeOverride : public boost::false_type {};
```

To provide customized evaluation strategies, this class must be specialized. For example, the specialization for fused vector addition is the following.

```
template<typename LhsType, typename Op, typename RhsType>
struct BinaryBinaryEvaluateNodeOverride<LhsType, Op, RhsType,
    typename boost::enable_if
    <
            NodeCanUnroll<expt::Node<LhsType, Op, RhsType> >
    >::type> : public boost::true_type
{
    template<typename ResultType, typename ArgumentVectorType>
    static inline void Evaluate(ResultType& accumulator, const
        ArgumentVectorType& args)
    {
        Unroll::Execute(accumulator, args);
    }
};
```

By inheriting from `boost::true_type`, we are indicating that we have provided an overload. In this case, the `NodeCanUnroll` metafunction returns true when the node represents a sequence of vector additions, and the `Unroll` class provides the implementation. Our library provides a similar override for BLAS functions to provide optimized evaluation when the expression matches functions that exist in BLAS.

These building blocks form the basis of the compile-time optimization of the parse tree. Evaluating an expression is done by first optimizing the parse tree, then

evaluating the optimized tree with an accumulator that traverses the tree in a depth-first fashion according to Algorithm 1.

With this framework in place, we can now evaluate an entire expression.

```
 1 template<typename Expression>
 2 static void Evaluate(const Expression& expression, typename Expression::
       ResultType& accum)
 3 {
 4     typedef typename Expression::Indices Indices;
 5
 6     // Perform the optimizations on the parse three.
 7     typedef typename RemoveUnecessaryTemporaries<Expression>::
           TransformedNodeType OptimizedParseTree;
 8     typedef typename RemoveUnecessaryTemporaries<Expression>::
           TransformedIndicesType TransformedIndicesType;
 9
10     if( ContainsAlias(expression, accum) )
11     {
12         typename Expression::ResultType temp = CreateFromTree<typename
               Expression::ResultType, OptimizedParseTree,
               TransformedIndicesType, 0>::Apply(expression.GetData());
13         EvaluateNode<OptimizedParseTree, TransformedIndicesType>::Evaluate(
               temp, expression.GetData());
14         accum = temp;
15      }
16     else
17     {
18         EvaluateNode<OptimizedParseTree, TransformedIndicesType>::Evaluate(
               accum, expression.GetData());
19     }
20 }
```

This method is meant to be called from within a constructor or assignment operator as follows.

```
class Matrix
{
    template<typename Expr>
    Matrix(const Expr& expr)
    {
        Evaluate(expr, *this);
    }

    template<typename Expr>
    Matrix& operator=(const Expr& expr)
    {
        Evaluate(expr, *this);
        return *this;
    }
};
```

The parameters to Evaluate (line 2) are the expression to be evaluated and the accumulator that will be used during parse tree evaluation. The expression that is passed to the class' copy constructor or assignment operator is the unoptimized parse tree that is created by stringing together the nodes from each operator in the expression

(as described in Section 3.1). Parse tree optimization is performed at compile time in lines 7 and 8. The RemoveUnecessaryTemporaries class is a metafunction that takes a parse tree as its template parameter and returns the optimized parse tree and the updated indices as member typedefs.

The next step is to determine if the accumulator is aliased in the expression, which occurs at line 10. This detects the aliasing in expressions such as $A = A + B$ and prevents an operand from being used as the accumulator. If an alias is detected, then a temporary is created to act as the accumulator during expression evaluation.

Expression evaluation occurs at lines 13 and 18. The evaluation uses the compile-time parse tree and the compile-time index calculations (as indicated by the template parameters). The runtime components representing the actual operands to the expression are passed as the argument to the function. The operands are still in the order written; the evaluator will use the indices to put them in the correct order.

## 6. PERFORMANCE

In this section, we demonstrate the effectiveness of our library using several different test cases. Timings were performed on an 8-core Intel Core i7 2.5 GHz machine running 64-bit Ubuntu 12.10 and 64-bit Windows 7. Tests were performed using gcc 4.7.2 and gcc 4.5.4 (with compile flags −O3), and Visual Studio 2010 (with compile flags /O2 /Ob2). All tests were executed using double-precision floating-point arithmetic and were executed in a single thread.

The goal of all of these tests is to determine if a particular expression template implementation provides a benefit when compared to a default, unoptimized implementation. In these tests, the unoptimized implementation is implemented using the Matrix and Vector objects that are part of Nektar++ in which the expression templates are not enabled. Each test implementation is then compared by calculating the speedup with the equation $t_b/t_e$, where $t_b$ is the unoptimized time, and $t_e$ is the time from the expression template library under test.

### 6.1. Vector Addition

For this test, we evaluate the performance of adding $n$ vectors:

$$v = \sum_{i}^{n} v_i. \tag{3}$$

All expression template libraries, including the one presented in this article, implement this in terms of loop fusion. Therefore, we do not expect any of the tests using expression templates to create temporary objects, and therefore total running times between implementations are expected to be approximately the same. This test illustrates that, even though the focus of this article has been the reduction of temporaries in an accumulator-based evaluation context, our library is also capable of evaluating trees using current best-practice approaches when applicable.

We performed this test with seven different implementations. The baseline for all tests was run using the Nektar++ Vector objects without an expression template library. For comparison, we ran the same test using the expression template frameworks from the MET [Ito 2001], UBlas [Walter and Koch 2002], Blitz++ [Veldhuizen 1998], and Eigen [Guennebaud et al. 2010] frameworks, as well as a hand-optimized version using loop fusion, and the same Nektar++ Vector objects combined with the expression template framework described in this aticle.

We note that our tests involved using the assignment operator rather than copy constructor since some frameworks (notably MET) were unable to construct vector objects from an expression.

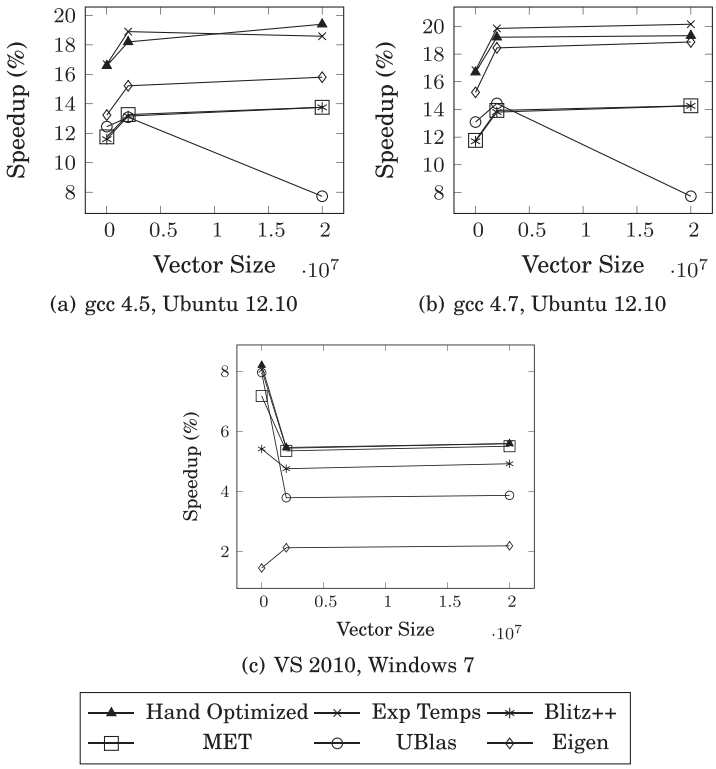



Fig. 9. Time to evaluate vector addition for a varying number of input vectors.

The results of these tests are shown in Figure 9. All of the expression template libraries performed similarly since all libraries implement a form of loop fusion for vector addition. They are also all significantly better than using C++ objects directly without any optimization.

### 6.2. Matrix Multiplication

This test evaluates the multiplication of linear algebra matrices:

$$M = \prod_{i}^{n} M_i. \tag{4}$$

We compared a C++ matrix multiplication expression with and without expression templates to the optimized BLAS routine DGEMM. Results are shown in Figure 10, as well as the Armadillo and Eigen expression template libraries. The other expression template libraries did not support chaining of matrix multiplication and, when we created workaround tests, had runtimes significantly slower than those presented here.

### 6.3. Set Operations

We can see from the previous sections that our expression template library is competitive with existing libraries when executing linear algebra-based operations. Where our library distinguishes itself is in its ability to be applied to contexts outside of linear algebra. In Figure 11, we show performance tests, with and without expression

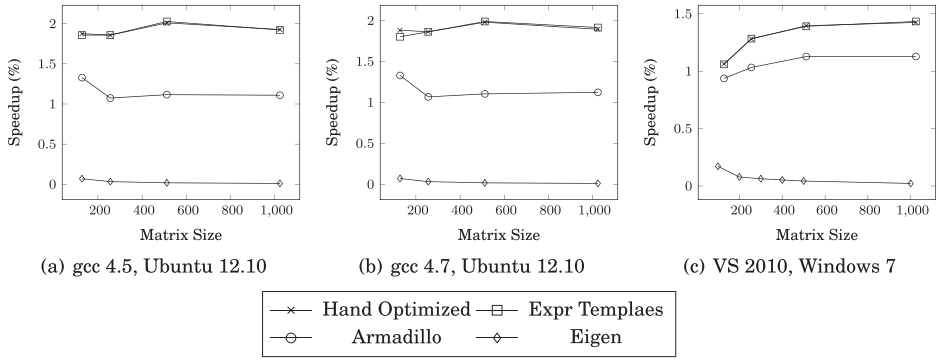



Fig. 10. Time to evaluate the chained matrix multiplication $m_0 * m_1 * m_2 * m_3$, in terms of speedup compared to the reference Nektar++ implementation without expression templates.

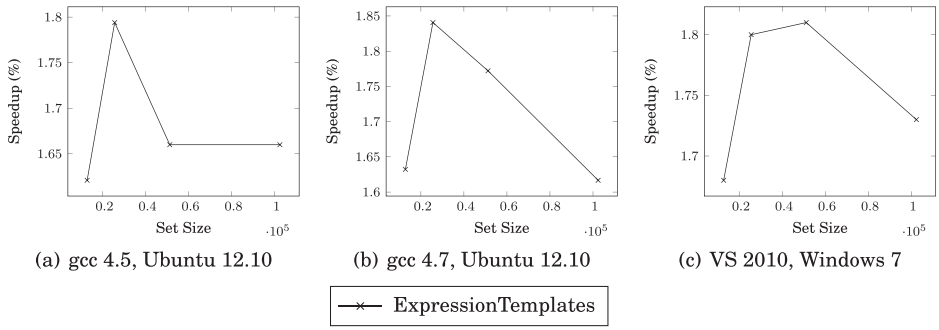
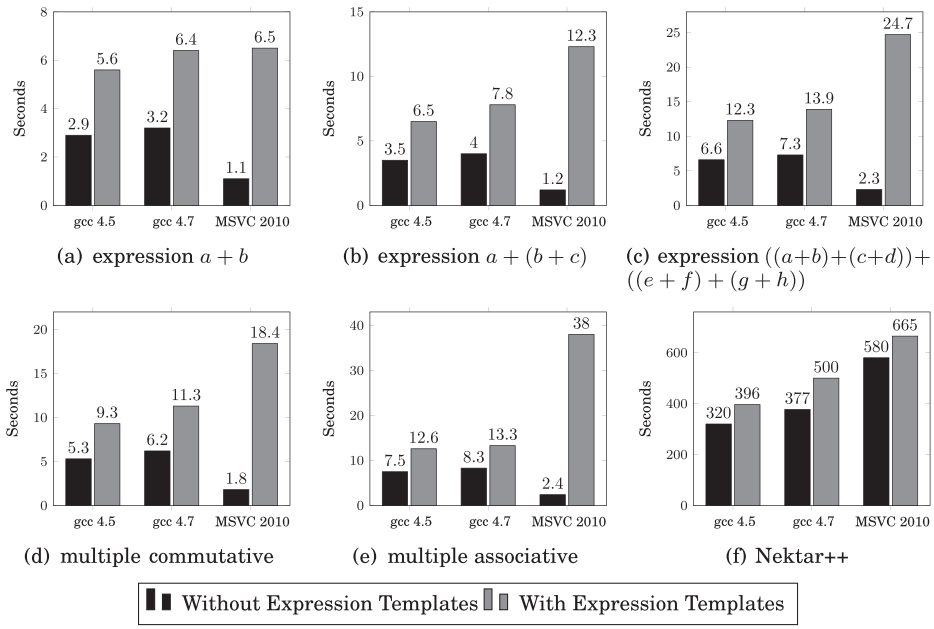



Fig. 11. Speedup of expression templates evaluating the set expression $(A \cup (B \cup C)) \cap A$ compared to a reference implementation that does not use expression templates.

templates, for the equation $(A \cup (B \cup C)) \cap A$. We see that with expression templates, the expression can be evaluated nearly twice as fast as without. Each set was the same size, with $A \cap B \cap C = \emptyset$. Both implementations used `std::set<unsigned int>` as the storage, `std::set_union` to implement union, and `std::set_intersection` to implement set intersection. What distinguishes this test from the previous tests is that the union and intersection operations do not lend themselves to loop fusion, so the accumulator-based approach presented in this work is a good fit.

### 6.4. Compilation Performance

Template metaprogramming is well known to increase compile times, sometimes significantly. An expression template library that increases compile time by too much will be of limited use. To evaluate the impact our library has on compilation times, we created test cases that consist of 2000 instantiations of a given expression. These expressions use objects that wrap floating-point values, allowing us to apply expression templates to floating-point operations. The result of these tests are shown in Figure 12(a)–(e). In Figure 12(a), we show performance for the expression $a + b$. In Figure 12(b), we show the performance for the expression $a + (b + c)$. We ran this test for scenarios where $+$ is neither commutative nor associative, where $+$ is associative, and where $+$ is commutative. These combinations were chosen to exercise the various transforms in our library. In all cases, the compilation time was the same, so we illustrate all cases in a single graph. In Figure 12(c), we tested $((a+b)+(c+d))+((e+f)+(g+h))$ to test a large associative cluster. In Figure 12(d)




(a) expression $a + b$ (b) expression $a + (b + c)$ (c) expression $((a+b)+(c+d))+((e + f) + (g + h))$

(d) multiple commutative (e) multiple associative (f) Nektar++

Fig. 12. Compilation performance for several test expressions, as well as when compiling the Nektar++ framework.

we tested $(a + bc) + (d + ef)$, and in 12(e), we tested an expression with multiple small associative clusters, $((a + (b + c)) * (d + (e + f))) + (g + (h + i))$.

We can see from these tests that compiling with expression templates is much more expensive than compiling without, especially for Visual Studio. However, these times are for a worst-case scenario of source files densely populated with expression evaluations. In a more typical case, expressions will be found distributed throughout the source code. To investigate the performance of this case, we compiled the Nektar++ project with and without expression templates. Results of this test can be found in Figure 12(f). Here, we see that while compile times did increase with expression templates, the increase is more manageable, ranging from 10 to 35 percent, depending on the compiler.

## 7. CONCLUSION

We have presented a new expression template framework that is capable of evaluating expressions using the minimum number of temporaries by manipulating the expression's parse tree at compile time. Using this framework, expression evaluation can be optimized by the compiler, allowing users to concentrate on the mathematical logic rather than the computer implementation. This is particularly useful when writing programs that handle linear algebra constructs because the alternative is often to write dense, difficult-to-comprehend code.

We note that, even though our library will evaluate expressions with the fewest number of temporaries, this does not always translate into a noticeable performance improvement. This occurs when the cost of creating a temporary is small compared to the cost of evaluating the operation. While there is generally no harm in using our expression template library, the potential gains will be application specific.

Additional optimization may be possible by considering additional operator properties. For example, if the $*$ operator is distributive, then the expression $ax + bx$ can be expressed as $(a + b) * x$, which could then be evaluated without any temporaries. We

did not pursue this property because it would require runtime checks to determine the difference between the expressions $ax + bx$ and $ax + by$, which would likely negate any performance improvements achieved by the temporary reduction. Similarly, there is potential for performance improvement by identifying equivalent subexpressions, such as $ab$ in $ab + c(d - ab)$, and evaluating them only once.

## REFERENCES


Prud'hom Christophe. 2006. A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations. *Sci. Program. 14,* 2, 81–110. http://portal.acm.org/citation.cfm?id=1376891.1376895.

Daniele Antonio Di Pietro and Alessandro Veneziani. 2009. Expression templates implementation of continuous and discontinuous galerkin methods. *Comput. Vis. TSci.* 12, 8, 421–436.

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

Masakatsu Ito. 2001. Matrix expression templates. http://met.sourceforge.net/.

George Em Karniadakis and Spencer J. Sherwin. 1999. *Spectral/Hp Element Methods for CFD*. Oxford University Press, New York.

Robert C. Kirby. 2003. A new look at expression templates for matrix computation. *Comput. Sci. Engin. 5*, 3, 66–70.

Charles L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw. 5,* 3, 308–323.

Michael Lehn, Alexander Stippler, and Karsten Urban. 2005. Flens — A flexible library for efficient numerical solutions. http://dml.cz/bitstream/handle/10338.dmlcz/700445/Equadiff_11-2005-2_55.pdf.

Laurent Plagne, Frank Hülsemann, Denis Barthou, and Julien Jaeger. 2009. Parallel expression template for large vectors. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'09)*. ACM Press, New York.

Conrad Sanderson. 2010. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Tech. rep. NICTA, Australia. http://arma.sourceforge.net/armadillo_nicta_2010.pdf.

Ravi Sethi and Jeffrey D. Ullman. 1970. The generation of optimal code for arithmetic expressions. *J. ACM 17,* 4, 715–728.

David Vandevoorde and Nicolai M. Josuttis. 2002. *C++ Templates*. Addison-Wesley Longman, Boston, MA.

Todd L. Veldhuizen. 1995. Expression templates. *C++ Rep. 7,* 5, 26–31. (Reprinted in *C++ Gems*, Stanley Lippman, Ed.)

Todd L. Veldhuizen. 1998. Arrays in blitz++. In *Proceedings of the 2nd International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*. Springer, 223–230. http://portal.acm.org/citation.cfm?id=646894.709708.

Joerg Walter and Mathias Koch. 2002. Boost basic linear algebra library (ublas) homepage. http://www.boost.org/libs/numeric/ublas/.


Received February 2012; revised February 2013; accepted August 2013