Expression Templates Revisited: A Performance Analysis of the Current ET Methodology

Klaus Iglberger⋆, Georg Hager†, Jan Treibig†, Ulrich Rüde∗ ⋆

⋆ Central Institute for Scientific Computing
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany

† Erlangen Regional Computing Center
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany

∗ Chair for System Simulation
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany

November 26, 2024

Abstract:

In the last decade, Expression Templates (ET) have gained a reputation as an efficient performance optimization tool for C++ codes. This reputation builds on several ET-based linear algebra frameworks focused on combining both elegant and high-performance C++ code. However, on closer examination the assumption that ETs are a performance optimization technique cannot be maintained. In this paper we demonstrate and explain the inability of current ET-based frameworks to deliver high performance for dense and sparse linear algebra operations, and introduce a new "smart" ET implementation that truly allows the combination of high performance code with the elegance and maintainability of a domain-specific language.

Keywords:

Expression Templates, Performance Optimization, High Performance Programming, Linear Algebra, Boost, *uBLAS*, *Blitz++*, *Blaze*

1 Introduction

Expression Templates (ETs) as originally introduced by Veldhuizen in 1995 [16, 17] are intended to be a "performance optimization for array-based operations." The general goal is to avoid the unnecessary creation of temporary objects during the evaluation of arithmetic expressions with overloaded operators in C++. Commonly demonstrated using simple $\mathcal{O}(n)$ array operations like additions, it achieves performance levels similar to hand-crafted C code while maintaining an elegant mathematical syntax. This success led to quick adoption in standard textbooks [19, 1], and ETs are thus widely accepted as *the* technique for high-performance array math in C++.

Widely known libraries that fully implement ET-based arithmetics are *Blitz++* [3], which was developed as "a C++ class library for scientific computing which provides performance on par with Fortran 77/90," and Boost *uBLAS* [6], which is part of the Boost project [4]. Both frameworks successfully use ET concepts to avoid the creation of temporaries. They provide fast array arithmetic and still (mostly) maintain an intuitive, mathematical syntax via C++ operators. Also, both frameworks extend the ET methodology to matrices and provide BLAS level 2 and 3 operations. In comparison to *Blitz++*, Boost *uBLAS* extends the idea of ETs to sparse vectors and matrices.

The starting point of this paper is an evaluation of the single-core (serial) performance of the *Blitz++* and Boost *uBLAS* libraries in the context of high performance computing (HPC). Although the idea of expression templates has a wider scope (they are, e.g., used for lambda expressions [5]), we focus on their performance aspect in the context of numerical libraries. Based on those results we will explain in detail why the current ET methodology is not suited for high performance computing in general. As a solution we describe an alternative ET approach, which combines the advantages of a high-level language with architecture-specific performance optimization, and is thus intrinsically suited for HPC. This "smart" ET methodology is implemented in the *Blaze* library that was developed in context of the *pe* physics engine [10]. Note that we ignore GPGPU computing altogether and focus on a contemporary CPU architecture, the Intel "Westmere." In order to demonstrate the achievable performance, we will compare all results from the ET libraries to optimized BLAS code (using the Intel MKL [11]).

The paper is organized as follows. In Section 2 we will give a short overview of related work, before Section 3 briefly summarizes the details of our benchmark platform. Section 4 recapitulates the current ET techniques and evaluates ET performance for the standard benchmark (dense vector addition). In Section 5 we extend the analysis to dense matrix-matrix multiplication and uncover some of the limitations of standard ETs. We turn to study the use of ETs for sparse data structures and complex expressions (operator chaining) in Sections 6 and 7. Finally we propose the new methodology of "Smart Expression Templates," which corrects the problems of standard ETs by combining the positive aspects of a domain-specific language with BLAS performance, in Section 8. Section 9 elaborates on the aspect of inlining in the context of ETs, before Section 10 concludes the paper and provides suggestions for future work.

2 Related Work

Not many groups have invested work to look into the performance of ETs. Bassetti [2] analyzed the performance of C++ expression templates in comparison to Fortran 77 code. They show that the promise of performance of ETs is not uniformly guaranteed across the different implementations of ETs, which they blame on the high demand on registers in complex ET implementations. Härdtlein [12] introduced the concept of "easy expression templates", which are easier to implement than classical ET, and the concept of "fast expression templates", which use static memory to improve the performance of array operations.

3 Benchmark Platform

A 6-core Intel Westmere CPU at 2.93 GHz with 12 MByte of shared L3 cache was used for all benchmarks. The GNU g++ 4.4.2 and Intel 11.1 compilers produced very similar performance results, so we always only present the results of the GNU g++ compiler. To allow a direct comparison of the different ET methodologies, we do not employ any low-level optimization apart from proper loop ordering, where appropriate. *Blitz++*, Boost *uBLAS*, and *Blaze* were benchmarked as given. All results are normalized to the fastest measured performance across the different frameworks for each particular test case. For all test cases with dense vectors and matrices we additionally provide MFlops/s values.

4 The Idea Behind Expression Templates

In this section we will recap the basic mechanisms at work in ETs. As an example, we will use the addition of two dense vectors of type `Vector`[1]:

[1] We will focus on the essential aspect of expression templates here and therefore omit all unnecessary details. For instance, we are aware that the `Vector` class could be implemented as a class template, but this would unnecessarily bloat the code

Listing 1: Addition of two dense vectors

```
Vector a, b, c;
// ... Initialization of vector a and b
c = a + b;
```

The use of the C++ arithmetic operators allows for a very concise description of the addition operation: The two vectors `a` and `b` are added and the result is assigned to the third vector `c`. Assuming that the `Vector` class allows access to its elements via the subscript operator and provides a `size` function to query its current size, `operator+` is usually implemented similar to the following code:

Listing 2: Classic implementation of the addition operator.

```
1 inline const Vector operator+( const Vector& a, const Vector& b )
2 {
3    Vector tmp( a.size() );
4 
5    for( size_t i=0; i<a.size(); ++i )
6       tmp[i] = a[i] + b[i];
7 
8    return tmp;
9 }
```

Although very intuitive to use and very flexible (it is for instance possible to concatenate vector additions), the performance of this implementation in comparison with hand-crafted C code exhibits bad performance due to the creation of the temporary vector `tmp` in line 6. The creation of `tmp` involves a dynamic memory allocation, a copy operation from the temporary into the target vector, and a memory deallocation. Additionally, the temporary interferes with cache locality due to the increased memory footprint of the operation. All this additional overhead, however, could be removed by implementing the vector addition manually:

Listing 3: C-like, manual implementation of the addition of two vectors.

```
for( size_t i=0; i<size; ++i )
   c[i] = a[i] + b[i];
```

The performance loss is even worse if several vectors are added within a single statement due to the "greedy" expression evaluation [1]:

Listing 4: Addition of three dense vectors.

```
Vector a, b, c, d;
// ... Initialization of vector a, b, and c
d = a + b + c;
```

For each single addition operation a separate temporary vector is created, whereas the operation would not require a single temporary:

Listing 5: C-like, manual implementation of the addition of three vectors.

```
for( size_t i=0; i<size; ++i )
   d[i] = a[i] + b[i] + c[i];
```

The ET approach is to create a compile-time parse tree of the whole expression to remove the creation of the costly temporary objects entirely and to delay the execution of the expression until it is assigned to its target. Therefore the addition operator no longer returns the (computationally expensive) result of the addition, but a small temporary object that acts as a placeholder for the addition expression [9]:

Listing 6: ET-based implementation of the addition operator.

```
template< typename A, typename B >
class Sum
{
 public:
  explicit Sum( const A& a, const B& b )
     : a_( a )
     , b_( b )
```

and obscure the core of ETs.

```
  {}

  std::size_t size() const {
    return a_.size();
  }

  double operator[]( std::size_t i ) const {
    return a_[i] + b_[i];
  }

 private:
  const A& a_;  // Reference to the left-hand side operand
  const B& b_;  // Reference to the right-hand side operand
};


template< typename A, typename B >
Sum<A,B> operator+( const A& a, const B& b )
{
  return Sum<A,B>( a, b );
}
```

Instead of calculating the result of the addition of two vectors, the addition operator now returns an object of type `Sum<A,B>`, where `A` and `B` are the types of the left- and right-hand side operands, respectively. The only requirements the addition operator poses on `A` and `B` are the existence of a subscript operator to access the elements of the operands and a `size` function. The `Sum` class has two data members, which are references-to-const to the two operands of the addition operation. Therefore this object is cheap to create and copy in comparison to the complete result vector. Since the `Sum` class represents the result of an addition, it must provide access to the resulting elements. For this purpose, it defines two access functions: The `size` function to access the size of the resulting vector and the subscript operator to access the individual elements.

The `Sum` class now temporarily represents the addition, until a special assignment operator is encountered:

Listing 7: Implementation of the ET assignment operator.

```
class Vector
{
 public:
  // ...

  template< typename A >
  Vector& operator=( const A& expr )
  {
    resize( expr.size() );

    for( std::size_t i=0; i<expr.size(); ++i )
      v_[i] = expr[i];

    return *this;
  }

  // ...
};
```

This assignment operator is the only other assignment operator of the `Vector` class next to the copy assignment operator (which is necessary in case of a manual management of the memory for the vector elements). Every time an expression object is assigned to a `Vector`, this assignment operator is used to handle the assignment[2]. It first resizes the vector accordingly and afterwards traverses the elements of the given expression within a single `for`-loop. Note that during this traversal the evaluation of the expression is triggered due to the access to the values via the subscript operator. Also note that this `for`-loop is the only `for`-loop necessary to evaluate the entire expression.

Via this formulation based on the inline formulation of all functions and the evaluation within a single `for`-loop hidden in the assignment operator the compiler is able to generate code similar to a C-like implementation (see Listing 3). It is even possible to concatenate several additions as for instance illustrated

[2] The thorough reader might notice that due to the signature of this assignment operator all non-vector objects assigned to a vector that do not fit the signature of the copy assignment operator will use this assignment operator. How this problem is handled is explained in detail in [9] and [10].

in Listing 4 without the creation of any temporary object (and still a single `for`-loop evaluation as in Listing 5).

Both the Boost *uBLAS* as well as the *Blitz++* library are based on the two major ideas of the illustrated ET implementation:

- no temporaries are created during the evaluation of an expression (except for the ET objects themselves, which also have to be considered temporaries)
- the elements of the left-hand side target are evaluated element-wise by the time the assignment operator is called and by accessing the elements of the right-hand side expression

In the following, we are comparing the performance of six different implementation of the addition of two dense vectors. The first contestant is the classic C++ operator overloading technique. Contestant number two is a C-like, manual implementation of the `for`-loop, as illustrated in Listing 3. The third approach is a plain function that accepts the two operands and the target vector of type `Vector` as arguments and wraps the vector addition:

Listing 8: Implementation of the addition of two vectors in a plain function.

```
inline void addVectors( const Vector& a, const Vector& b, Vector& c )
{
   // ... Same implementation as in Listing 2, except no temporary is created
}
```

Contestants four and five are the *Blitz++* and Boost *uBLAS* libraries, respectively. The sixth contestant is the *Blaze* library that will be introduced in Section 8.

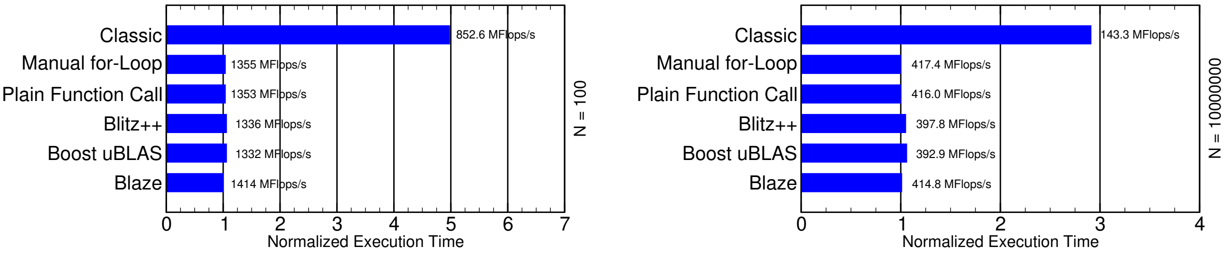

Figure 1: Performance comparison between six different implementations of the addition between two dense vectors.

Figure 1 shows the performance results for both small vectors (in-cache) and large vectors (out-of-cache). As expected, the classic C++ operator overloading shows by far the worst performance due to the extra data transfer caused by the temporary vector. In this direct comparison it becomes obvious that the overhead due to the creation of a temporary vector prevents good performance. In this regard, ETs can be considered a performance optimization in comparison to naive C++ operator overloading: By avoiding the creation of an intermediate temporary, they achieve the performance of a manual, C-like implementation of the vector addition. Additionally, they provide the expressiveness, naturalness, and flexibility of a domain specific language [1] by exploiting operator overloading, i.e. it is for instance possible to intuitively concatenate the addition of several vectors.

5 ETs: A Performance Optimization Technique?

The reputation that ETs are a performance optimization exclusively results from their performance advantage compared to classic C++ operator overloading in BLAS level 1 operations, such as the dense vector addition. No further performance comparisons have been published so far. One main reason for that is that the optimization of array operations is, according to Veldhuizen's original publication, the main application of ETs. Still, both *Blitz++* as well as Boost *uBLAS* provide functionality well beyond the BLAS level 1 operations, which, according to the *Blitz++* homepage, also "[...] provides performance on par with Fortran 77/90". In this section, we will evaluate the performance of a BLAS level 3 function, the multiplication between two dense matrices. The characteristics of the dense matrix multiplication make this operation a particularly well suited candidate for optimization, since with a proper optimization (memory access scheme, etc.) this operation can be made arithmetically bound instead of memory bound [8].

For this comparison we use six different implementations of a plain multiplication between two dense matrices. The first implementation is a straight forward C++ implementation using the classic operator overloading technique. Listing 9 shows the according implementation that, except for a suited ordering of the nested `for`-loops does not contain any optimizations.

Listing 9: Implementation of the matrix-matrix multiplication operator.

```cpp
inline const Matrix operator*( const Matrix& A, const Matrix& B )
{
   Matrix C( A.rows(), B.columns() );

   for( size_t i=0; i<A.rows(); ++i ) {
      for( size_t k=0; k<B.columns(); ++k ) {
         C(i,k) = A(i,0) * B(0,k);
      }
      for( size_t j=1; j<A.columns(); ++j ) {
         for( size_t k=0; k<B.columns(); ++k ) {
            C(i,k) += A(i,j) * B(j,k);
         }
      }
   }

   return C;
}
```

The second contestant is the implementation of a plain function accepting the three involved matrices as arguments. This function is similar to the `addVector` function from Listing 8. The third and fourth contestant are the *Blitz++* (see Listing 10) and Boost *uBLAS* (see Listing 11) libraries, respectively. The fifth implementation is provided by the *Blaze* library (see Listing 12) and the sixth code uses a plain call to the `dgemm` BLAS function.

Listing 10: Use of the matrix multiplication in the *Blitz++* library.

```cpp
blitz::Array<double,2> A( N, N ), B( N, N ), C( N, N );
blitz::firstIndex i;
blitz::secondIndex j;
blitz::thirdIndex k;
// ... Initialization of the matrices
C = blitz::sum( A(i,k) * B(k,j), k );
```

Listing 11: Use of the matrix multiplication in the Boost *uBLAS* library.

```cpp
boost::numeric::ublas::matrix<double> A( N, N ), B( N, N ), C( N, N );
// ... Initialization of the matrices
noalias( C ) = prod( A, B );
```

Listing 12: Use of the matrix multiplication in the *Blaze* library.

```cpp
pe::MatN A( N, N ), B( N, N ), C( N, N );
// ... Initialization of the matrices
C = A * B;
```

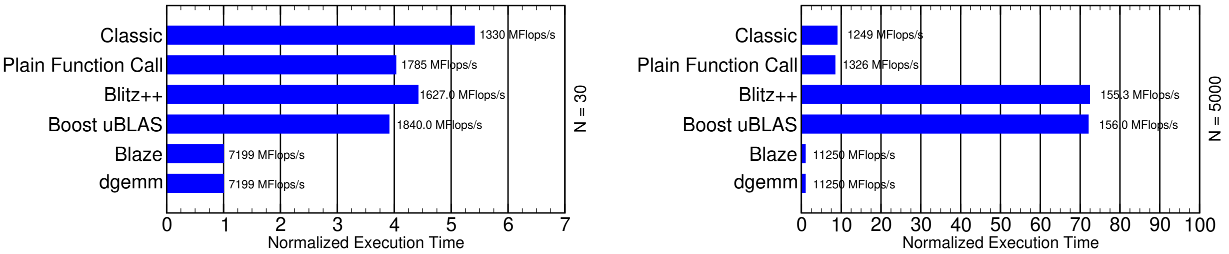

Figure 2: Performance comparison between five different implementations of the multiplication between two dense matrices.

Figure 2 shows the performance results for the six different implementations. For both an in-cache matrix multiplication with two matrices of size 30^2 as well as the out-of-cache multiplication with two

		Memory Bandwidth [MBytes/s]	Total Retired Instructions [10^{11}]	Total Arithmetic Operations [10^{11}]	Cycles Per Instruction (CPI)
	STREAM	11814	—	—	—
N=5000	Classic	5008	12.5054	2.50231	0.441127
	Plain Function Call	5328	12.5048	2.50232	0.440912
	Blitz++	623	10.0126	2.58185	4.67952
	Boost *uBLAS*	623	10.0053	2.50197	4.72096
	Blaze	496	2.02589	2.50612	0.322074
	dgemm	496	2.02589	2.50612	0.322074

Table 1: Likwid performance analysis of the multiplication between two dense matrices. Note that the `dgemm` function uses packed instructions, which may results in a higher number of arithmetic operations than retired instructions!

matrices of size 5000^2 the `dgemm` function is clearly the fastest competitor. However, although also based on ETs, the *Blaze* library achieves the same performance level, since internally the *Blaze* also uses the `dgemm` function (see Section 8). In contrast, the other two ET-based libraries exhibit very poor performance. Whereas this result comes with no real surprise, since the purpose of the `dgemm` function is to provide a maximum of performance for the matrix multiplication, the fact that even the simple, non-optimized, old fashioned operator overloading performs much better in case of the out-of-cache matrices than the ET-based libraries is surprising.

Table 1 gives an indication of why the performance of Boost *uBLAS* and *Blitz++* is so bad. We used the Likwid tool suite [18] to measure the achieved memory bandwidth, the total number of retired instructions, the total number of arithmetic operations, and the number of cycles per instruction (CPI). When comparing the CPI both Boost *uBLAS* as well as *Blitz++* reveal a low quality of the generated code. In combination with the high number of retired instructions and the low achieved memory bandwidth it becomes obvious why the used ET implementation is shows low performance.

The reason for this behavior is intrinsic to the methodology of ETs. Based on the philosophy that each element of the target data structure is computed one after another, the executed code is similar to the code shown in Listing 13:

Listing 13: Slow implementation of the matrix-matrix multiplication operator.

```
for( size_t i=0; i<A.rows(); ++i ) {
   for( size_t j=0; j<B.columns(); ++j ) {
      for( size_t k=0; k<A.columns(); ++k ) {
         C(i,j) += A(i,k) * B(k,j);
      }
   }
}
```

This loop ordering corresponds to the worst possible data access scheme that can be used for the matrix multiplication: For each element of the target matrix a complete column of the right-hand side matrix is traversed, resulting in a cache line transfer for each individual data value. Especially for out-of-cache matrices, this approach is very cache inefficient since only a single value of each cache line can be used before the cache line has to be replaced. In contrast to this loop ordering, the two codes for classic operator overloading and the plain function call use a more cache efficient data access scheme that simultaneously calculates several values of the target matrix, which results in a much better memory bandwidth and lower CPI.

Although in case of the classic operator overloading technique a temporary is created, which is omitted in case of the ET libraries, the performance of the classic technique is much better. Obviously, the performance gain results from the choice of the better data access scheme. Thus the primary question is why the ET libraries don't implement the more efficient loop ordering in order to gain performance. The reason for this is that with the current methodology ETs are not able to choose the best data access scheme. ETs are solely based on the goal to avoid temporaries, the strategy to evaluate the given right-hand side expressions element-wise, and the firm believe that the compiler will optimize the resulting code constructs after inlining took place. Whereas this works well for array operations as for instance the vector addition, where there is barely opportunity for data access scheme optimization and where therefore the omission of the temporary results in a performance optimization, in order to achieve high performance for the matrix multiplication the detailed knowledge about the involved data structures and the operation has to be exploited.

The fundamental problem of the current ET technique is that it is no performance optimization technique, but essentially an abstraction technique. Whereas this abstraction improves the flexibility of a framework to integrate new types and operations, it counteracts high performance on several levels. First, ETs abstract from the involved data types. A clear indication for this is that the involved ET data types are required to adhere to a certain interface ("Design by Contract" [15]). Therefore no special optimization can be applied based on the type of the used matrices. Second, ETs abstract from the type of operation. From an abstract point of view it makes no difference whether the target matrix is assigned a matrix addition expression or a matrix multiplication expression; In both cases, the according assignment operator accesses the elements of this virtual matrix to fill the target matrix. However, in terms of performance a matrix addition has to be treated fundamentally different from a matrix multiplication. Therefore, with the current methodology, real performance optimization based on memory optimization (the most important optimization for contemporary, cache-based architectures [8]), vectorization, and exploitation of superscalarity, cannot be properly performed. The optimization capability of ETs is thus limited to operations where the abstract data access scheme coincidentally corresponds to the optimal data access scheme.

These results have another important implication. A crucial aspect of ETs is the encapsulation of the numerical operations in functions. By this they provide an intuitive, easy-to-use interface and a very high maintainability. This aspect is especially important for complex numerical operations, such as the matrix multiplication: Whereas simple numerical kernels, such as a vector addition, can easily be rewritten, it should not be necessary to repeatedly reimplement complex kernels, which contain a huge amount of work to achieve high performance. From a performance point of view, the encapsulation of complex kernels is therefore more important than the encapsulation of simple kernels. Considering the performance results for the matrix multiplication, it must be concluded that the current ET methodology is not suitable to encapsulate highly optimized complex kernels.

6 Sparse Arithmetic

Due to the abstraction from the actual data types in all operations, ETs offer an impressive flexibility to integrate new data types into the system. The abstraction is achieved by requiring all data types to adhere to a certain interface via which it is possible to access the underlying elements. One example, for what this flexibility can be used, is demonstrated by the Boost *uBLAS* library: In contrast to *Blitz++*, Boost *uBLAS* provides sparse vectors and matrices that can be homogeneously combined with the available dense vectors and matrices. This enriched functionality is clearly an extraordinary strength of ETs. The downside of this abstraction, however, is a performance penalty. In order to show this performance penalty, we selected two operations between dense and sparse data types and compared their performance between Boost *uBLAS* and the *Blaze*.

The first operation is the multiplication between a row-wise stored sparse matrix and a dense vector. This type of operation is of importance in many engineering applications as it is for instance used to solve linear systems of equations. Listing 14 shows its implementation with the Boost *uBLAS* library, Listing 15 with the *Blaze* library.

Listing 14: Use of the sparse matrix/dense vector multiplication in the Boost *uBLAS* library.

```
boost::numeric::ublas::compressed_matrix<double> A( N, N );
boost::numeric::ublas::vector<double> a( N ), b( N );
// ... Initialization of the matrix and the vectors
noalias( b ) = prod( A, a );
```

Listing 15: Use of the sparse matrix/dense vector multiplication in the *Blaze* library.

```
SparseMatrixMxN<double> A( N, N );
Vector<double> a( N ), b( N );
// ... Initialization of the matrix and the vectors
b = A * a;
```

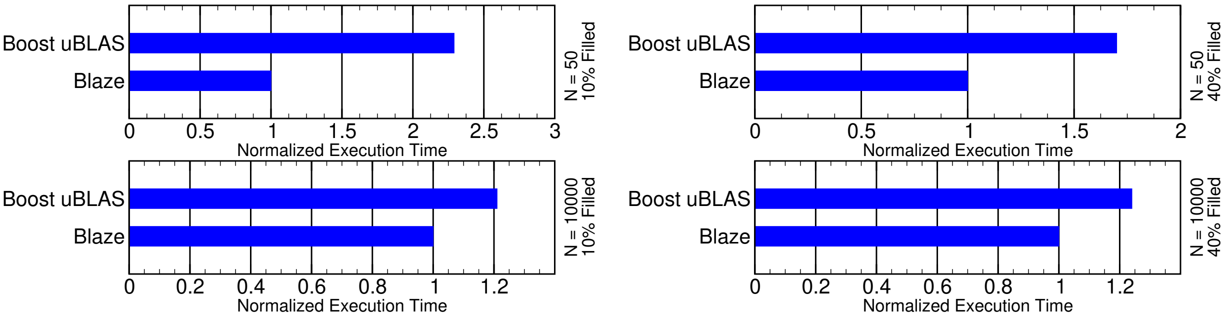

Figure 3: Performance comparison between Boost *uBLAS* and *Blaze* for the multiplication between a sparse matrix and a dense vector.

Figure 3 shows the in-cache and out-of-cache performance results for a 10% and 40% filled sparse matrix, respectively. The direct comparison between Boost *uBLAS* and the *Blaze* does not exhibit a huge performance difference neither for the different sizes nor the different filling degrees. The reason for that is that the default memory access scheme utilized by the ET implementations works perfectly for this operation: A single row of the matrix has to be multiplied with the dense vector for each result vector element. Since both the row-wise memory access to the sparse matrix as well as the access to the dense vector perfectly exploit the structure of both data structures, the performance is on a reasonable level.

Listing 16: Use of the dense matrix/sparse matrix multiplication in the Boost *uBLAS* library.

```
boost::numeric::ublas::matrix<double> A( N, N ), C( N, N );
boost::numeric::ublas::compressed_matrix<double> B( N, N );
// ... Initialization of the matrix and the vectors
noalias( C ) = prod( A, B );
```

Listing 17: Use of the dense matrix/sparse matrix multiplication in the *Blaze* library.

```
MatrixMxN<double> A( N, N ), C( N, N );
SparseMatrixMxN<double> B( N, N );
// ... Initialization of the matrices
C = A * B;
```

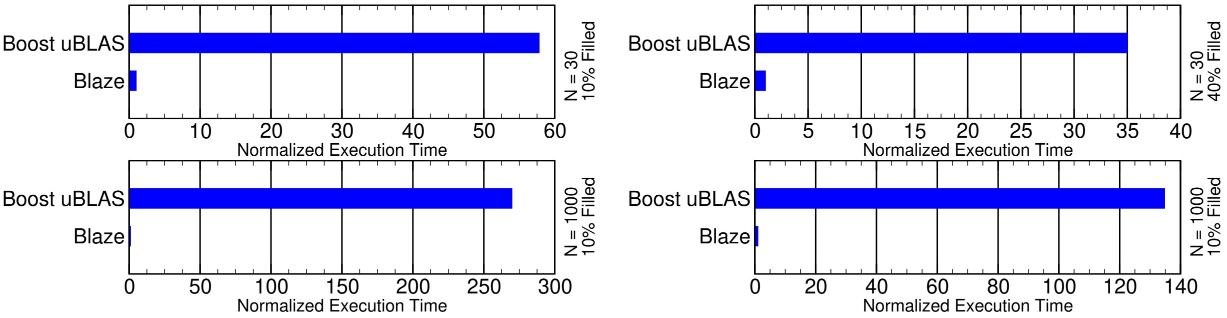

Figure 4: Performance comparison between Boost *uBLAS* and *Blaze* for the multiplication between a dense and a sparse matrix.

The situation changes entirely when we multiply a row-wise stored dense matrix with a row-wise stored sparse matrix. Listing 16 shows the implementation of this operation with the Boost *uBLAS* library, Listing 17 shows its implementation with the *Blaze*. Figure 4 shows the in-cache and out-of-cache performance results for 10% and 40% filled sparse matrices, respectively. It becomes obvious that there is a tremendous performance difference between the two libraries that cannot be explained by simple differences in the implementation of the codes, but points at fundamental differences in the methodology of the two ET libraries. Whereas the *Blaze* attempts to exploit all information about the operations and both data types and therefore deals as efficiently as possible with the fact that the right-hand side sparse matrix is stored in a row-wise fashion, Boost *uBLAS* completely abstracts from the current operation and the data types of the two involved matrices. All elements of the result matrix are evaluated one after another [3] by traversing the left-hand side dense matrix via row-iterators and the right-hand side sparse matrix via column iterators. Although the column iterators can be considered a very convenient interface for users of the library, their internal, abstract use results in a devastating performance penalty in this case. What would be required instead in order to achieve high performance would be a recognition of the data structure of the right-hand side sparse matrix and the attempt to use and reuse its elements in a cache-efficient manner. However, due to the abstraction from both the actual operation as well as the data types, this is not possible. Therefore the current methodology of ETs prohibits any possible performance optimization for this operation.

Note that this operation was specifically selected to demonstrate that performance greatly suffers from the abstraction from the data types and operations. The performance penalty would be much less severe in case of a column-wise stored sparse matrix. However, since ET libraries are usually provided as black box systems, the knowledge that the combination of certain data structures should be (completely) avoided, cannot be expected from a user of the library.

7 Complex Expressions

One of the two fundamental rules of ETs is that no temporaries are created during the evaluation of an expression in order to avoid all overhead involved in creating a temporary. This rule is mainly responsible for the reputation that ETs are a performance optimization. However, in certain situations the creation of a temporary is strictly necessary to achieve performance although it involves extra work. In this section we have specifically selected two examples for complex expressions that require the creation of temporaries in order to demonstrate the shortcoming of this rule.

The first complex expression is the multiplication between a dense matrix and the sum of three dense vectors: $A \cdot (a + b + c)$. The problem that is involved in this expression is obvious: The right-hand side vector of the matrix-vector multiplication is required several times during its evaluation. In case the result of the vector additions $a + b + c$ is not computed prior to the multiplication, the additions have to be evaluated several times, which will inevitably result in a performance loss.

Listing 18: Use of the expression $d = A * (a + b + c)$ with classic operator overloading.

```
classic::Matrix<double> A( N, N );
classic::Vector<double> a( N ), b( N ), c( N ), d( N );
// ... Initialization of the matrix and vectors
d = A * ( a + b + c );
```

Listing 19: Use of the expression $d = A * (a + b + c)$ in the *Blitz++* library.

```
blitz::Array<real,2> A( N, N );
blitz::Array<real,1> a( N ), b( N ), c( N ), d( N ), tmp( N );
blitz::firstIndex i;
blitz::secondIndex j;
// ... Initialization of the matrix and vectors
tmp = a + b + c;
d   = blitz::sum( A(i,j) * tmp(j), j );
```

Listing 20: Use of the expression $d = A * (a + b + c)$ in the Boost *uBLAS* library.

```
boost::numeric::ublas::matrix<real> A( N, N );
boost::numeric::ublas::vector<real> a( N ), b( N ), c( N ), d( N );
// ... Initialization of the matrices
noalias( d ) = prod( A, ( a + b + c ) );
```

[3] As a reminder: This approach is necessary due to the abstraction from the actual operation!

Listing 21: Use of the expression $d = A * (a + b + c)$ in the *Blaze* library.

```
pe::MatrixMxN<double> A( N, N );
pe::VectorN<double> a( N ), b( N ), c( N ), d( N );
// ... Initialization of the matrices
d = A * ( a + b + c );
```

Listings 18, 19, 20, and 21 show the implementation of the complex expression with classic operator overloading, *Blitz++*, Boost *uBLAS* and *Blaze*, respectively. Interestingly, it is necessary to explicitly create the temporary `tmp` in case of *Blitz++* since it is syntactically not possible to evaluate the complex expression within a single statement. Figure 5 shows the in-case and out-of-cache performance results of four different implementations of this expression: classic operator overloading, the *Blitz++* library, Boost *uBLAS*, and the *Blaze* library.

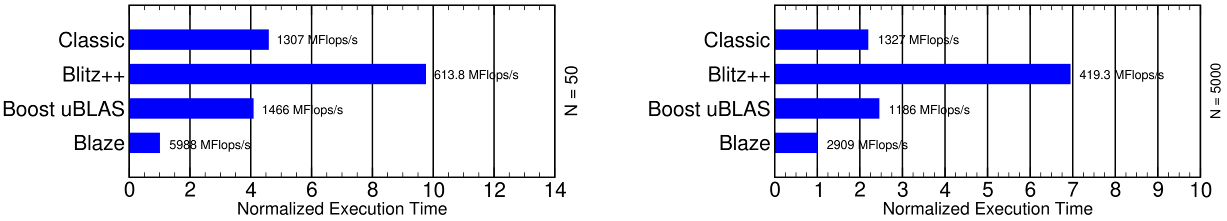

Figure 5: Performance comparison between four different implementations of the complex expression $A \cdot (a + b + c)$.

For both small and large N, the two traditional, ET-based libraries do not exhibit good performance. Especially in case of large N, classic operator overloading, although requiring a total of three temporaries for the evaluation of the expression, performs better than Boost *uBLAS* and especially *Blitz++*. The *Blaze* library, which uses a single temporary to store the intermediate result of the vector additions and utilizes the optimized `dgemv` function for the subsequent matrix-vector multiplication, has a clear performance advantage. Table 2 shows the Likwid results, which allow to analyze the performance results in more detail. It becomes obvious that the better cache utilization as well as the memory bandwidth of the optimized `dgemv` function result in higher performance.

		Memory Bandwidth [MBytes/s]	Total Retired Instructions [10^8]	Total Arithmetic Operations [10^7]	Cycles Per Instruction (CPI)	L1 Data Cache Line Replacements [10^6]
	STREAM	11814	—	—	—	—
N=5000	Classic	5387	4.31892	7.56438	0.455758	6.32184
	Blitz++	2295	6.87758	7.55893	0.531862	6.36924
	Boost *uBLAS*	4382	4.5681	12.5684	0.529004	12.5812
	Blaze	11088	2.74818	7.74858	0.57686	3.99694

Table 2: Likwid performance analysis of the complex expression $A \cdot (a + b + c)$.

The second complex expression we selected involves four dense matrices: $E = (A + B) \cdot (C - D)$. In this case, in order to efficiently be able to evaluate the matrix multiplication, both the left-hand side as well as the right-hand side matrix expression must be evaluated prior to the matrix multiplication. Again, in case of *Blitz++*, the expression cannot be computed within a single statement, which results in two explicit temporary matrices. The benefit of this can be seen in Figure 6, which shows both the in-cache as well as the out-of-cache results for classic operator overloading, *Blitz++*, Boost *uBLAS*, and the *Blaze* library. *Blitz++* always performs better than Boost *uBLAS*, which does not create any intermediate temporaries and re-evaluates the matrix addition and subtraction repeatedly. However, both *Blitz++* as well as Boost *uBLAS* exhibit poor performance in comparison to the *Blaze* library, which internally creates two temporaries to hold the intermediate results of the matrix addition and subtraction and uses the `dgemm` function to compute the subsequent matrix multiplication. Especially striking is the fact that for large N both *Blitz++* and Boost *uBLAS* are extremely by classic operator overloading, since it does create the necessary temporaries and utilizes a faster kernel for the matrix multiplication. These performance are confirmed by the Likwid results in Table 3. Based on the large data cache replacement rate, the large CPI and the low memory bandwidth the quality of the generated code is poor.

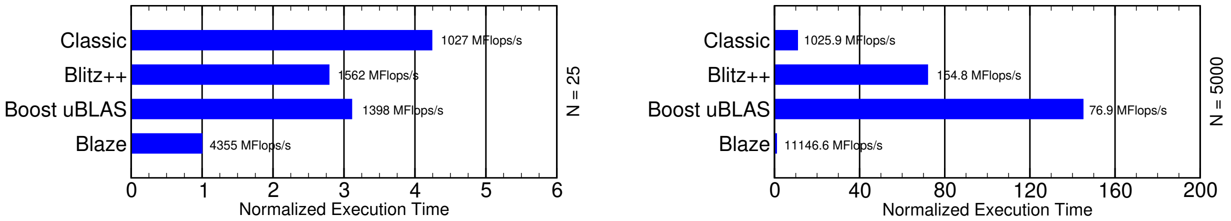

Figure 6: Performance comparison between four different implementations of the complex expression $(A + B) \cdot (C - D)$.

		Memory Bandwidth [MBytes/s]	Total Retired Instructions $[10^{11}]$	Total Arithmetic Operations $[10^{11}]$	Cycles Per Instruction (CPI)	L1 Data Cache Line Replacements $[10^{9}]$
	STREAM	11814	—	—	—	—
N=5000	Classic	4136	12.5106	2.50318	0.442167	31.3006
	Blitz++	624	10.0163	2.56789	4.68541	266.566
	Boost *uBLAS*	619	13.7553	6.15386	6.90581	533.411
	Blaze	490	2.02977	2.50684	0.322925	2.07864

Table 3: Likwid performance analysis of the complex expression $(A + B) \cdot (C - D)$.

Admittedly, a simple solution to improve the performance of *Blitz++* and Boost *uBLAS* would be the explicit generation of temporaries whenever necessary. This solution is also for instance advertised on the Boost *uBLAS* homepage, where the *uBLAS* designers advocate the reintroduction of temporaries as a performance remedy[4]. However, arguably the primary goals of ETs is the ability to use infix operator notion and to provide a convenient, intuitive black box interface for all kinds of mathematical operations. Therefore a user of these libraries cannot be blamed for the lack of proper automatic recognition of necessary temporaries. Since these libraries provide this interface, they have to expect that someone actually uses the interface and therefore have to take care of all possible consequences, including the requirement to create automatic temporaries. The fundamental ET rule to avoid all temporaries, which established the reputation of ETs being a performance optimization, can therefore obviously also act as performance “pessimization”.

8 A New ET Methodology: Smart Expression Templates

Obviously ETs themselves are not generally able to provide high performance. However, the idea to combine high performance code with the mathematical syntax provided by the C++ operators is a justified one: code clarity, readability, and maintainability are greatly improved if used in a mathematical context.

In this section we will shortly sketch the smart ET methodology of the *Blaze* library by means of the multiplication between two dense matrices. In contrast to other ET frameworks the ET implementation of the *Blaze* library takes a completely different approach in order to achieve the goal of combining performance and syntax. In *Blaze* the notion of ETs being a performance optimization is completely dropped. The ETs merely act as a parsing functionality that understands the structure of the given mathematical expression, knows the order in which subexpressions need to be evaluated (including the creation of temporaries; see Section 7)[5], and selects the appropriate, highly optimized kernels, which provide a manual, architecture-specific performance optimization based on detailed knowledge of data types and operations.

Note that this section serves only as a cursory overview of the methodology of smart expression templates. We will not go into detail of the very sophisticated C++ implementation; The implementation along with a detailed discussion will be published in a separate article. Instead, we only try to explain

[4] In fact, the suggested solutions often involve ungraceful syntactical expressions that don’t have anything left from the elegance ETs try to achieve

[5] An illustrating example for a smart choice for the evaluation order of subexpressions is the expression $A*B*v$, where A and B are two matrices and v is a vector. Usually the expression is evaluated from left to right, resulting in a matrix-matrix multiplication and a subsequent matrix-vector multiplication. However, if the right subexpression were be evaluated first, the performance can be dramatically improved since the matrix-matrix multiplication can be avoided in favor of a second matrix-vector multiplication.

the idea of the smart ET methodology by focusing on the two key concepts: the selective creation of intermediate temporaries and the integration of optimized kernels.

8.1 Creation of Intermediate Temporaries

The first key idea of smart ETs involves the creation of intermediate temporaries. The fundamental rule of ETs not to create temporaries has two different reasons. The first reason is the abstraction from the actual operations. Due to that the necessity to create a temporary cannot be recognized. The second reason seems to be the apparent impossibility to efficiently create intermediate temporaries from subexpressions inside an expression object, since the creation of a temporary is always associated with a memory allocation, a copy operation, and a memory deallocation.

The solution for this problem is already incorporated in the C++ standard itself. Consider the following operation:

Listing 22: Addition of three dense vectors.

```
Vector a, b;
// Initialization of vector a and b
Vector c = a + b;  // Same as: Vector c( a + b );
```

In contrast to the dense vector addition shown in Listing 1, in this case we do not perform an assignment, but rather an initialization of the dense vector `c`. Therefore there is no performance difference between all implementations: Even classic operator overloading exhibits the same performance as the ET-based libraries. The reason behind this is the “named return value” (NRV) optimization (see section 12.1.1c of the ARM [7] or [14]). Listing 23 shows the compiler-optimized implementation of the addition operator from Listing 2. If the compiler applies NRV to a code (which is triggered by the presence of an explicit copy constructor), the local variable `tmp` will be replaced by a reference to the eventual destination of the return value in the caller and instead of returning a temporary the function returns `void`.

Listing 23: NRV optimization of the dense vector addition operator.

```
inline void operator+( Vector& dest, const Vector& lhs, const Vector& rhs )
{
  dest.Vector::Vector( lhs.size() );  // Explicit constructor call

  for( std::size_t i=0; i<lhs.size(); ++i )
    dest[i] = lhs[i] + rhs[i];
}
```

In case of an initialization, the compiler can therefore directly write the result to the destination vector, which corresponds to the behavior that is achieved by the ET formulation. In case of an assignment, a temporary is created by means of the NRV optimized code, which is in turn assigned to the destination vector:

Listing 24: Compiler generated code for the copy assignment of vectors

```
Vector a, b, c;

// NRV optimized addition of a and b into the temporary tmp
Vector tmp( a + b );

// Assignment of the temporary to the vector c
c = tmp;
```

In the context of ETs, in case an intermediate temporary is created (for instance as the result of a subexpression), it is created via initialization (not assignment). The same is true for the creation of the temporary expression objects themselves. Hence the creation of temporaries does not involve a single copy operation, but only the necessary memory allocations and deallocations. Therefore in the smart ET methodology temporary objects are used to hold intermediate results of subexpressions as member variables of other expression objects.

8.2 Integration of Optimized Compute Kernels

The second key idea of smart expression templates is the selection of the appropriate compute kernel. The solution is to omit the abstract assignment via the assignment operator, by passing this responsibility to the resulting expression object. Since the expression object holds all knowledge about the involved data types and operations it can perform the assignment as efficiently as possible. The following code excerpt shows how this optimization is implemented in case of the `DMatDMatMultExpr` class that represents the multiplication between two dense matrices:

Listing 25: Smart expression object for the matrix-matrix multiplication

```
template< typename MT1    // Type of the left-hand side dense matrix
        , typename MT2 >  // Type of the right-hand side dense matrix
class DMatDMatMultExpr : private Expression
{
 public:
  // Public interface omitted

 private:
  // ...

  // Result type of the left-hand side dense matrix expression
  typedef typename MT1::ResultType     RT1;

  // Result type of the right-hand side dense matrix expression
  typedef typename MT2::ResultType     RT2;

  // Composite type of the left-hand side dense matrix expression
  typedef typename MT1::CompositeType  CT1;

  // Composite type of the right-hand side dense matrix expression
  typedef typename MT2::CompositeType  CT2;

  // Member data type of the left-hand side dense matrix expression.
  typedef typename SelectType<IsExpression<MT1>::value,const RT1,CT1>::Type  Lhs;

  // Member data type of the right-hand side dense matrix expression.
  typedef typename SelectType<IsExpression<MT2>::value,const RT2,CT2>::Type  Rhs;

  Lhs lhs_;  // Left-hand side dense matrix of the multiplication expression.
  Rhs rhs_;  // Right-hand side dense matrix of the multiplication expression.

  // Specialized assign function injected into the surrounding namespace
  template< typename MT >  // Type of the target dense matrix
  friend inline void assign( DenseMatrix<MT>& lhs,
                             const DMatDMatMultExpr& rhs )
  {
    // Depending on the data type utilization of the cblas_dgemm kernel or
    // use of the default implementation of the matrix-matrix multiplication
  }

  // ...
};
```

The `DMatDMatMultExpr` is implemented as a template parameterized with the two data types `MT1` and `MT2` of the involved dense matrices. The class is derived from the `Expression` class, which makes the `DMatDMatMultExpr` class an expression (in contrast to plain matrices). Via "Template Meta Programming" (TMP) [1] the data types of the two operands are used to evaluate the two member data types `Lhs` and `Rhs`. In case any of these types is an expression (i.e., derived from the `Expression` class), the `ResultType` of the according matrix expression is used to create a temporary object (optimized by the NRV optimization and therefore without a copy operation). Otherwise the `CompositeType` of the matrix expression is used, which represents the knowledge of the expression how it should be treated in a composite expression.

The core of the class is the `assign` function, which implements the assignment of the matrix-matrix multiplication to a dense matrix. This function is injected into the surrounding namespace via the Barton-Nackman trick [13, 19]. In case of an assignment of a temporary `DMatDMatMultExpr` object to a dense matrix this function is called, which performs the assignment of the matrix multiplication based on the fastest available compute kernel. Depending on the types of the matrix operands it either applies a default matrix multiplication kernel (which works with any data type) or a call to the optimized BLAS functions (`cblas_sgemm` for single-precision matrices and `cblas_dgemm` for double-precision matrices).

In summary, the smart ET methodology of the *Blaze* considers ETs as an intelligent wrapper technology around a collection of highly optimized kernels that provide operation, data type and architecture specific optimizations. A remarkable advantage of this methodology is that this kernel based approach allows an easy integration of multi- and many-core optimizations as well as GPU-based kernels.

9 Inlining

Inlining is an essential issue for all ET-based frameworks: Without a complete inlining of the entire ET functionality the expected performance level cannot be achieved. Therefore ETs are vitally depending on the inlining capabilities of the used compiler. However, due to the enormous number of nested function calls in ET codes the pressure on the compiler is very high. Additionally, the `inline` keyword is merely a recommendation for the compiler to perform the inlining and not a binding instruction. Depending on the size of the function the ETs are used in, the size of the compilation unit, the total number of instructions, etc. the compiler might reject this recommendation and choose to insert function calls.

During our performance measurements we frequently encountered problems with failed inlining, even within apparently small test programs to measure the performance of a certain operation. Therefore inlining seems to be a real issue that might result in bad performance although the implementation would be able to deliver much more. In our measurements, we went to great lengths to ensure that all ET functionality was properly inlined to measure the maximum possible performance. In order to demonstrate the impact of failed inlining, however, Figure 7 shows a comparison between proper and failed inlining in case of the dense vector addition (the inlined performance values correspond to the results from Figure 1).

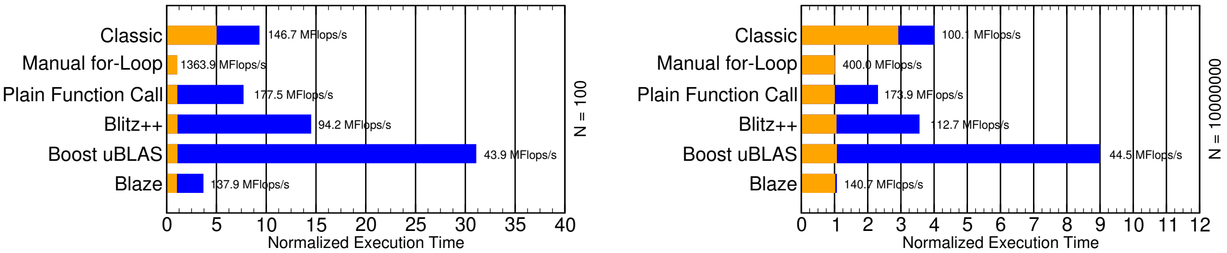

Figure 7: Performance comparison of the dense vector addition for proper and failed inlining.

As this comparison shows, inlining poses a severe and fundamental problem for all ET-based production code[6]. Most importantly, programmers must not be overly confident in the compiler's ability to (1) perform inlining to the required level and then (2) generate the most efficient low-level loop code possible.

10 Conclusion and Future Work

There is very little ground for the reputation of standard Expression Templates to be a performance optimization for array operations. They do achieve their original goal of providing fast element-by-element array arithmetic in combination with the benefits of high-level constructs, because they effectively eliminate the generation of temporaries in expressions. In this sense, they remedy a specific deficiency of the C++ language. However, more complex operations like BLAS level 2 and 3 procedures, sparse linear algebra, and generally everything that profits from standard and architecture-specific low-level optimizations, often show devastating performance levels. This is because ETs are essentially an abstraction technique that hides the details of actual data and operations types and reduces them to efficient single-element access, which is insufficient: We have shown that the widespread belief in advanced inlining and optimization capabilities of C++ compilers is naive and unjustified. While aggressive inlining is a necessary prerequisite for getting good performance from ET source, it does not guarantee best low-level code. There is no replacement for exploiting all possible knowledge about data types, operations, and access patterns.

We have also introduced a new ET methodology, which we call "Smart Expression Templates." It eliminates the shortcomings of standard ETs by reducing the ET mechanism to an intelligent wrapper around a selection of highly optimized kernels or, in case of BLAS-type operations, vendor-provided libraries. Smart ETs combine the advantages of a domain-specific language (ease of use by high-level constructs, readability, encapsulation, maintainability) with the performance of HPC-suitable code. Moreover, they do not rely on aggressive inlining as much as standard ETs do.

In this work we have restricted our discussion to sequential code. Considering the importance of highly hierarchical, multicore/multisocket building blocks in today's high performance systems, a generalization of smart ETs to parallel computing on distributed data structures seems natural and will be investigated.

[6] The authors have to admit that this also affects the ET implementation of the *Blaze* library, but due to the concept of embedding HPC-kernels much less than the other ET frameworks.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming.* C++ In-Depth Series. Addison-Wesley, 2005.

[2] F. Bassetti, K. Davis, and D. Quinlan. C++ Expression Templates Performance Issues in Scientific Computing. In *Parallel Processing Symposium '98*, 1998.

[3] Blitz++ library. Homepage of the Blitz++ library: http://www.oonumerics.org/blitz/.

[4] Boost. Homepage of the Boost C++ framework: http://www.boost.org.

[5] Boost Lambda library. Homepage of the Boost Lambda library: http://www.boost.org/doc/libs/1_45_0/doc/html/lambda.html.

[6] Boost uBLAS library. Homepage of the Boost uBLAS library: http://www.boost.org/doc/libs/1_45_0/libs/numeric/ublas/doc/index.htm.

[7] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[8] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* Chapman & Hall/CRC Computational Science Series. CRC Press, 2010.

[9] J. Härdtlein. *Moderne Expression Templates Programmierung - Weiterentwickelte Techniken und deren Einsatz zur Lösung partieller Differentialgleichungen.* PhD thesis, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2007.

[10] K. Iglberger. *Software Design of a Massively Parallel Rigid Body Framework.* PhD thesis, 2010.

[11] Intel Math Kernel Library (MKL). Homepage of the IMKL framework: http://www.intel.com/software/products/mkl.

[12] J. Härdtlein and C. Pflaum and A. Linke and C. H. Wolters. Advanced Expression Template Programming. *Computing and Visualization in Science*, 13(2):59–68, 2009.

[13] J. J. Barton and L. R. Nackman. Algebra for C++ Operators. *C++ Report*, 7(3):70–74, 1995.

[14] S.B. Lippman. *Inside the C++ Object Model.* Addison-Wesley, 10th printing edition, 2007.

[15] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1997.

[16] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, 1995.

[17] T. Veldhuizen. Expression Templates. In *C++ Gems*, pages 475–487. SIGS Publications, Inc., New York, NY, USA, 1996.

[18] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI2010)*, 2010.

[19] D. Vandevoorde and N.M. Josuttis. *C++ Templates - The Complete Guide.* Addison-Wesley, 2003.