Expression Templates Revisited: A Performance Analysis of Current ET Methodologies

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

August 25, 2011

Running Head: ETs Revisited: A Performance Analysis

Corresponding Author:

Klaus Iglberger
Central Institute for Scientific Computing
Friedrich-Alexander University of Erlangen-Nuremberg
Martensstr. 5a
91058 Erlangen
Germany
klaus.iglberger@zisc.uni-erlangen.de

Co-Authors:

Georg Hager
Erlangen Regional Computing Center
Friedrich-Alexander University of Erlangen-Nuremberg
Martensstr. 3
91058 Erlangen
Germany
georg.hager@rrze.uni-erlangen.de

Jan Treibig
Erlangen Regional Computing Center
Friedrich-Alexander University of Erlangen-Nuremberg
Martensstr. 3
91058 Erlangen
Germany
jan.treibig@rrze.uni-erlangen.de

Ulrich Rüde
Chair for System Simulation
Friedrich-Alexander University of Erlangen-Nuremberg
Cauerstr. 6
91058 Erlangen
Germany
ulrich.ruede@informatik.uni-erlangen.de

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 compare the performance of several generations of ET-based frameworks. We analyze different ET methodologies and explain the inability of some ET implementations to deliver high performance for dense and sparse linear algebra operations. Additionally, we introduce the notion of "smart" ETs, which truly allow for a 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++*, *MTL4*, *Eigen3*, *Blaze*

1 Introduction

Expression Templates (ETs) as originally introduced by Veldhuizen in 1995 [21, 22] 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 [24, 1], and ETs are thus widely accepted as *the* technique for high-performance array math in C++.

Two widely known libraries that use the standard ET concepts to fully implement ET-based arithmetics are *Blitz++* [4], which was developed as "a C++ class library for scientific computing which provides performance on par with Fortran 77/90," and Boost *uBLAS* [7], which is part of the Boost project [5]. Both frameworks successfully use ETs 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* further extends the idea of ETs to sparse vectors and matrices.

Two representatives of a new generation of ET-based linear algebra libraries are *MTL4* [20] and *Eigen3* [8], both of which provide operations with dense vectors and dense and sparse matrices. Both introduce new concepts to improve the performance of standard ETs. *MTL4* uses BLAS function calls internally for certain calculations "to provide an easy and intuitive interface to users while enabling optimal performance." [20] In contrast, the technology of *Eigen3* is similar to standard ETs, but integrates explicit SIMD (single instruction multiple data) vectorization, exploits knowledge about fixed-size matrices, and implements standard unrolling and blocking techniques for better cache and register reuse.

The starting point of this work is an evaluation of the single-core (serial) performance of the *Blitz++*, Boost *uBLAS*, *MTL4*, and *Eigen3* 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 [6]), we focus on their performance aspect in the context of numerical libraries, whose performance is of fundamental importance. Based on those results we will explain in detail why the standard ET methodology is not suited for high performance computing in general and why simple extensions of it are still limited in the achievable performance. As a solution we propose the "smart" expression template (SET) concept. It offers the advantages of a high-level language, but overcomes the performance issues of standard expression templates by a combination of architecture-specific performance optimization, restructuring of expressions, and specific temporary creation, and is thus intrinsically suited for HPC. 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 library [14]).

This 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 standard ET techniques and evaluates ET performance for the basic 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. Section 8 elaborates on the aspect of inlining in the context of ETs. Finally we propose the new methodology of "Smart Expression Templates" in Section 9, before Section 10 concludes the paper and provides suggestions for future work.

2 Related Work

Few groups have invested work to look into the performance of ETs. Bassetti et al. [3] have analyzed the performance of C++ expression templates in comparison to Fortran 77 code. They show that the performance promise of ETs is not uniformly guaranteed across different implementations, which is attributed to the large register pressure in complex ET code. Härdtlein et al. [16] have introduced the concepts of "easy expression templates", which are easier to implement than standard ETs, and "fast expression templates", which use static memory to improve the performance of array operations.

3 Benchmark Platform

A six-core Intel Westmere CPU at 2.93 GHz with 12 MByte of shared L3 cache was used for all benchmarks. The maximum achievable memory bandwidth (as measured by the STREAM benchmark [18]) is about 20 GByte/s per socket. Note that this maximum cannot be hit by a single thread. Each core has a theoretical peak performance of 11.72 GFlop/s in double precision (DP), which can only be achieved if SIMD (Single Instruction Multiple Data) instructions are used. SIMD allows the execution of two arithmetic operations in one instruction (four in single precision), and is crucial for getting good performance in cache. Note that also the loads and stores to the memory hierarchy must be SIMD vectorized for best results.

The GNU g++ 4.4.2 and Intel 11.1 compilers produced very similar performance results, so we stick to GNU g++. The following compiler flags were used:

Listing 1: GCC compilation flags

```
g++ -Wall -Wshadow -Woverloaded-virtual -ansi -O3 -msse3 -DNDEBUG
    --param large-unit-insns=100000000 --param inline-unit-growth=100000000
    --param max-inline-insns-single=100000000 --param large-function-growth=100000000
    --param large-function-insns=100000000
```

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. We use the latest *Blitz++* implementation, Boost *uBLAS* in version 1.46, *MTL4* in version 4.0.8368, and *Eigen3* in version 3.0.1. All libraries were benchmarked as given. We only present double precision results either as MFlop/s graphs or normalized to the fastest measured performance across the different frameworks for each particular test case. However, we always provide MFlop/s values where possible. For all in-cache benchmarks we make sure that the data has already been loaded to the cache.

4 The Idea Behind Expression Templates

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

Listing 2: Addition of two dense vectors

```
1 Vector a, b, c;
2 // ... Initialization of vector a and b
3 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 3: 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 intuitive to use and flexible (it is for instance possible to concatenate vector additions), the performance of this implementation in comparison with hand-crafted C code is quite bad due to the creation of the temporary tmp in line 3. The creation of tmp involves a dynamic memory allocation, a subsequent copy operation from the temporary into the target vector (line 3 in Listing 2), and a memory deallocation. Additionally, the temporary interferes with cache locality due to the increased memory

[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 and obscure the core of ETs.

footprint of the operation. All this additional overhead, however, could be removed by implementing the vector addition manually:

Listing 4: 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 5: 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 expression would not require a single temporary:

Listing 6: 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 [12]:

Listing 7: 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 )
  {}

  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 8: Implementation of the ET assignment operator.

```cpp
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 due to the 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.

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 4). It is even possible to concatenate several additions as, e.g., illustrated in Listing 5, without the creation of any temporary object (and still a single **for** loop evaluation as in Listing 6).

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 eight implementations of the addition of two dense vectors:

1. Classic C++ operator overloading.
2. A C-like, manual implementation of the **for** loop, as illustrated in Listing 4.
3. A plain function that accepts the two operands and the target vector of type Vector as arguments and wraps the vector addition:

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

4. The *Blitz++* library.
5. The Boost *uBLAS* library.
6. The *MTL4* library.

[2]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 [12] and [13].

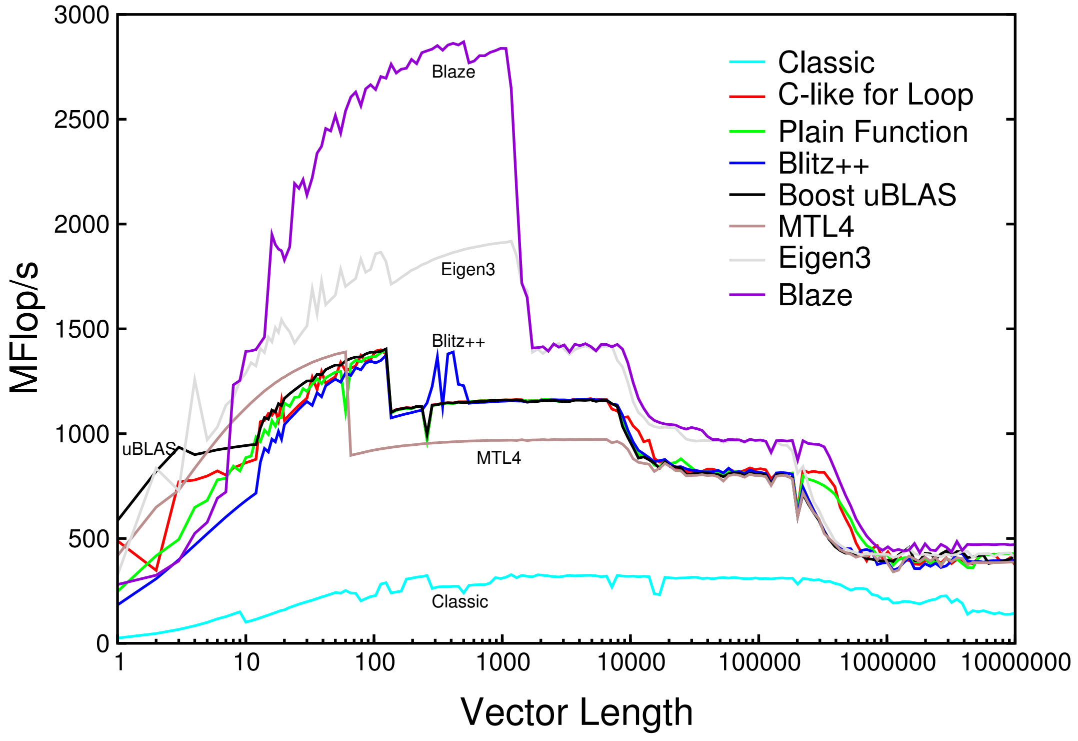

Figure 1: Performance comparison between eight implementations of the addition between two dense vectors.

7. The *Eigen3* library.
8. The *Blaze* library that incorporates SETs and which will be introduced in Section 9.

Figure 1 shows the performance results for vector sizes ranging from 1 to 10000000. 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 apparent that the overhead due to the creation of a temporary vector prevents good performance. In contrast, ET libraries such as *Blitz++*, Boost *uBLAS*, and *MTL4* using the just described standard approach and avoiding the creation of a temporary are able to achieve the performance of a manual, C-like implementation. In this regard, ETs can be considered a performance optimization in comparison to naive C++ operator overloading. 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.

However, a plain C loop is not the best that can be achieved on the given hardware. For small vector sizes the explicit use of SIMD intrinsics or compiler-based vectorization can improve the performance by about 30%, as can be seen from the results of *Eigen3* and *Blaze*. In case of large vectors, where the performance is limited by memory bandwidth, SIMD vectorization does not pay off, but the use of non-temporal stores can improve performance by roughly 20% in comparison to a C-like **for** loop. Non-temporal stores write data from registers directly to memory without using the cache hierarchy, eliminating the need for a write-allocate transfer on a write miss [11]. The memory traffic for the vector add operation is thus reduced by 25%. This result clearly demonstrates that compilers are usually not able to achieve the highest level of performance automatically and that extra effort is necessary to boost performance even for trivial operations such as the vector addition. In case of the *Eigen3* library the performance is achieved by relying on the optimization and inlining capabilities of the compiler, and by additionally providing access functions to packed data types. In contrast, the *Blaze* library ships with a specifically optimized dense vector addition kernel function, which limits performance for very small vectors, but yields best performance for all other vector sizes and is independent of compiler optimizations.

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. One main reason for that is that the optimization of array operations is the main application of ETs [21]. Still, *Blitz++*, Boost *uBLAS*, *MTL4*, and *Eigen3* provide functionality well beyond BLAS level 1. In this section we will evaluate the performance of a BLAS level 3 function, the multiplication of two dense matrices. The characteristics of the dense matrix multiplication make it an interesting optimization target, since it can be rendered arithmetically bound instead of memory bound via standard code transformations [11].

For this comparison we use eight implementations of a plain multiplication of two dense, row-major matrices:

1. A straightforward C++ implementation using classic C++ operator overloading. The following Listing shows the according implementation, which does not contain any optimizations except for a suitable ordering of the nested **for** loops .

```
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;
}
```

2. A plain function accepting the three involved matrices as arguments. This function is similar to the addVector function from Listing 3.

3. The *Blitz++* library.

```
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 );
```

4. The Boost *uBLAS* library.

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

5. The *MTL4* library.

```
mtl::dense2D<double> A( N, N ), B( N, N ), C( N, N );
// ... Initialization of the matrices
C = A * B;
```

6. The *Eigen3* library.

```
using Eigen::Dynamic;
using Eigen::RowMajor;

Eigen::Matrix<double,Dynamic,Dynamic,RowMajor> A( N, N ), B( N, N ), C( N, N );
// ... Initialization of the matrices
C.noalias() = A * B;
```

7. The *Blaze* library.

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

8. A plain call to the dgemm BLAS function

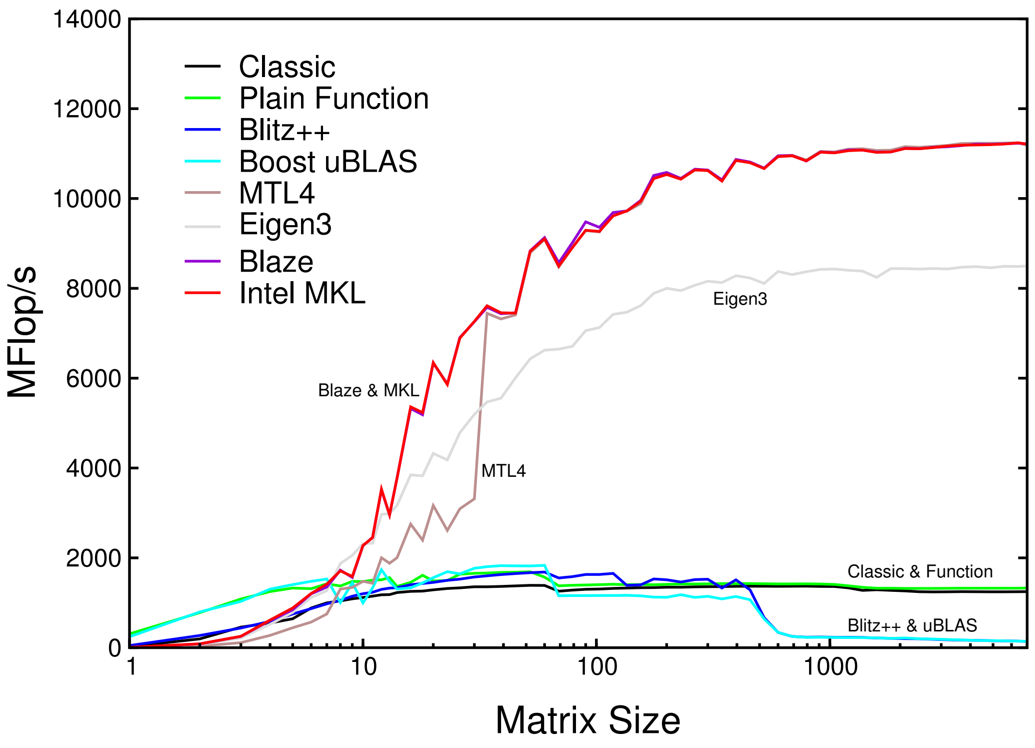

Figure 2: Performance comparison between eight implementations of the multiplication of two dense, row-major matrices.

Figure 2 shows the performance results for the eight implementations. For very small matrices, the overhead of the dgemm function call limits its achievable performance. However, for both small, in-cache and large, out-of-cache matrices the dgemm function is clearly the fastest option. However, although also based on ETs, the *MTL4* and the *Blaze* library achieve the same performance level, since internally they also use the dgemm function (see Section 9). In contrast, *Blitz++* and Boost *uBLAS* exhibit very poor performance. This result comes as no surprise, since the dgemm function in the MKL library is optimized for maximum performance. However, the fact that even the simple, non-optimized operator overloading is much faster for out-of-cache matrices than the ET-based libraries is unexpected.

Table 1 gives an indication of why the performance of Boost *uBLAS* and *Blitz++* is so low. We used the Likwid tool suite [23] to measure the following basic and derived hardware performance metrics when running the 5000^2 benchmark case:

- *Memory bandwidth.* The actual memory traffic caused by the operation (number of cache lines transferred to and from main memory multiplied by the cache line length of 64 bytes), divided by the runtime. For reference we have included the single-thread STREAM triad bandwidth, which constitutes a practical upper limit.
- *Total retired instructions.* The number of machine instructions that the processor has executed during the whole operation. It does not include instructions that were executed speculatively and later discarded due to, e.g., branch mispredictions.
- *Total arithmetic operations.* The number of floating-point operations (multiply or add in this case) executed. This may be larger than the number of retired instructions due to the use of SIMD vectorization. E.g., a double precision SIMD add instruction constitutes two floating-point operations.

		Memory Bandwidth [MByte/s]	Total Retired Instructions [10^{11}]	Total Arithmetic Operations [10^{11}]	Cycles Per Instruction (CPI)
	STREAM	11814	—	—	—
N=5000	Classic	5314	12.5420	2.50272	0.440861
	Plain Function Call	5318	12.5409	2.50270	0.440592
	Blitz++	633	10.1297	2.586	4.58243
	Boost *uBLAS*	630	10.1207	2.50349	4.61834
	MTL4	531	2.03452	2.50604	0.321143
	Eigen3	371	2.1014	2.53904	0.41168
	Blaze	531	2.03449	2.50604	0.321114
	dgemm	531	2.03448	2.50604	0.321115

Table 1: Hardware counter performance analysis of the multiplication of two large dense matrices ($N = 5000$). Note that the dgemm function uses packed instructions, which may result in a higher number of arithmetic operations than retired instructions. See the text for a description of all metrics.

- *Cycles per instruction (CPI).* The average number of cycles a machine instruction takes to execute. For the Westmere processor used here, the absolute maximum CPI as given by the architecture is 0.25, which will never be achieved by real application code. A value between 0.3 and 0.5 is considered "good," but it does not bear any information about whether "useful" code was executed. Hence, the CPI must always be assessed together with other, complementary metrics like the total number of retired instructions.

Looking at the memory bandwidth for the "classic" and "plain function call" codes, the main reason for their failure to deliver good performance is the lack of basic optimizations like (outer) loop unrolling and blocking, resulting in a considerable pressure on the memory interface (but still far away from the maximum). All other versions are all but decoupled from main memory, but differ considerably in other respects. Boost *uBLAS* and *Blitz++*, on the other hand, show a large number of retired instructions in combination with a very large CPI value, indicating a low-quality machine code, which is dominated by latencies.

The reason for this behavior is inherent to the methodology of standard 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 9:

Listing 9: 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 full cache line transfer for each individual data value. On the other hand, 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.

We must conclude that the temporary required for the classic operator overloading does not hurt here, since the cost is proportional to N^2. Thus the performance gain results from the choice of the better data access scheme. Thus the primary question is why the standard ET libraries do not implement a more efficient loop ordering. The reason is that they are solely based on three paradigms: (i) avoiding temporaries, (ii) evaluating the given right-hand side expressions element-wise, and (iii) relying on the compiler to perfectly optimize the resulting code constructs after inlining has taken place. Whereas this works reasonably well for array operations like the vector addition, in order to achieve high performance for the matrix multiplication detailed knowledge about the involved data structures, the operation, and the underlying hardware has to be exploited.

The fundamental problem of the standard ET method is that it is an abstraction technique rather than a performance optimization. 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" [19]). 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 in a fundamentally different way. Therefore, with the standard methodology, real performance optimization based on memory optimization (the most important optimization for contemporary, cache-based architectures [11]), 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.

That the ability of the compiler to optimize kernels is sometimes overestimated can be seen when analyzing the results of the *Eigen3* library. Although it is not suffering from the problems of standard ETs, the *Eigen3* library still does not achieve the performance level of an optimized dgemm function. While not abstracting from the right-hand side matrix operation and implementing a dense matrix multiplication kernel, *Eigen3* still relies on the compiler to assemble an optimized kernel from kernel building blocks. Therefore the performance of the *Eigen3* implementation is limited by the compiler's ability to optimize compute kernels and strongly depends on proper inlining (see Section 8).

These results have another important implication. A crucial aspect of ETs is the encapsulation of the numerical operations in functions, through which they provide an intuitive, easy-to-use interface and good maintainability. This aspect is especially important for complex numerical operations, such as the matrix multiplication: While simple kernels, such as a vector addition, can easily be rewritten (although it is by no means trivial to write a vector addition code that gives best performance under all circumstances), it should not be necessary to repeatedly reimplement complex kernels, which would require vast efforts: Hardware vendors have already invested into this, and usually provide suitable libraries. 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 standard 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 of this flexibility is demonstrated by the Boost *uBLAS*, *MTL4*, and *Eigen3* libraries: In contrast to *Blitz++*, they provide sparse data structures that can be homogeneously combined with the available dense vectors and matrices. This enriched functionality is an extraordinary strength of ETs. The downside of this abstraction, however, can be a severe performance penalty. In order to show this penalty we select two operations between dense and sparse data types and compare their performance.

Listing 10: Use of the sparse matrix/dense vector multiplication in the Boost *uBLAS* library.

```cpp
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 11: Use of the sparse matrix/dense vector multiplication in the *MTL4* library.

```cpp
typedef mtl::dense_vector<double>  dense_vector;
typedef mtl::tag::row_major  row_major;
typedef mtl::matrix::parameters<row_major>  parameters;
typedef mtl::compressed2D<double,parameters>  compressed2D;

compressed2D A( N, N );
dense_vector a( N ), b( N );
// ... Initialization of the matrix and the vectors
b = A * a;
```

Listing 12: Use of the sparse matrix/dense vector multiplication in the *Eigen3* library.

```
using Eigen::Dynamic;
using Eigen::RowMajor;

Eigen::SparseMatrix<double,RowMajor,size_t> A( N, N );
Eigen::Matrix<double,Dynamic,1> a( N ), b( N );
// ... Initialization of the matrix and the vectors
b.noalias() = A * a;
```

Listing 13: Use of the sparse matrix/dense vector multiplication in the *Blaze* library.

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

The first operation is the multiplication of a sparse matrix with a dense vector. It is of importance in many engineering applications, e.g., in solvers for linear systems of equations. Listings 10 through 13 show its implementation with the Boost *uBLAS*, the *MTL4*, the *Eigen3*, and the *Blaze* library, respectively. The matrix is stored in the well-known compressed row storage (CRS) format [2].

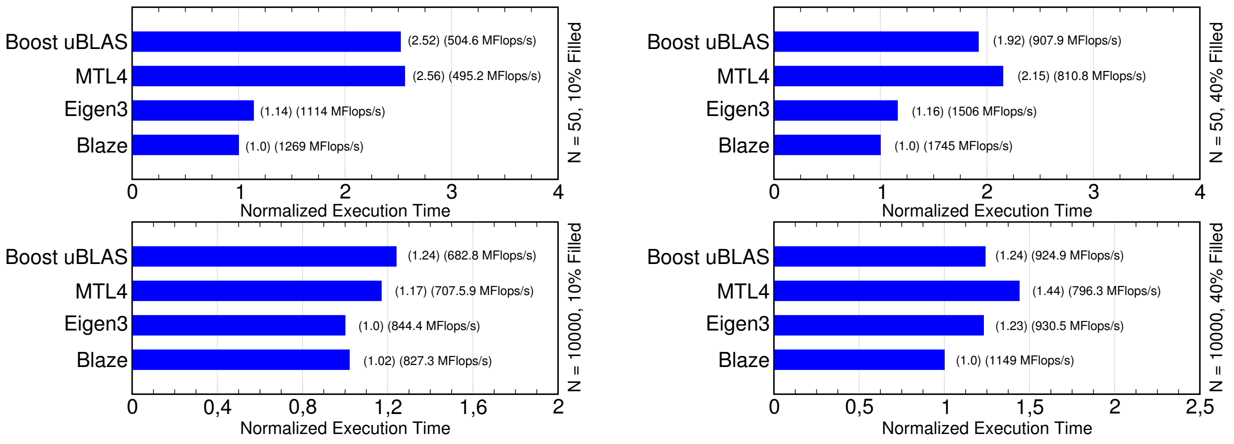

Figure 3: Performance comparison between four implementations of the sparse matrix by dense vector multiplication. In-cache vs. out-of-cache (top vs. bottom row) and different nonzero filling (left vs. right column) are distinguished.

Figure 3 shows the in-cache and out-of-cache performance results for sparse matrices randomly filled with 10% and 40% nonzeros, respectively. The direct comparison between the different ET implementations 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 element of the resulting vector. Since both the row-wise memory access to the sparse matrix as well as the access to the dense vector exploit the layout of both data structures, the performance is on a reasonable level with the exception of Boost *uBLAS* and *MTL4* for small matrices.

Similar performance levels can be expected for the multiplication of a row-wise stored dense matrix with a column-wise stored sparse matrix. However, the situation changes entirely when we multiply a row-wise stored dense matrix with a row-wise stored sparse matrix. Listings 14 through 17 show the according implementation of this operation with the Boost *uBLAS*, the *MTL4*, the *Eigen3*, and the *Blaze* library. Figure 4 shows the in-cache and out-of-cache performance results for 10% and 40% filled sparse matrices, respectively. There is a tremendous performance difference between the libraries that cannot be explained by simple variations in the implementation of the codes, but points at fundamental differences in the methodology of the ET libraries. Whereas *Blaze* attempts to exploit all information about the operations and both data types and therefore deal efficiently with the fact that the right-hand side sparse matrix is stored in a row-wise fashion, Boost *uBLAS* and *MTL4* completely abstract from the current operation and the layout and data types of the two involved matrices. In case of Boost *uBLAS*, all elements of the resulting matrix are evaluated one after another 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 are a convenient interface for users of the library, their internal, abstract use results in a devastating performance penalty in this case. To correct this would require a recognition of the data structure of the right-hand side sparse matrix and the use and reuse of 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 standard ETs prohibit any possible performance optimization for this operation.

Listing 14: 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 15: Use of the dense matrix/sparse matrix multiplication in the *MTL4* library.

```
typedef mtl::dense2D<double>  dense2D;
typedef mtl::tag::row_major  row_major;
typedef mtl::matrix::parameters<row_major>  parameters;
typedef mtl::compressed2D<double,parameters>  compressed2D;

dense2D A( N, N ), C( N, N );
compressed2D B( N, N );
// ... Initialization of the matrix and the vectors
C = A * B;
```

Listing 16: Use of the dense matrix/sparse matrix multiplication in the *Eigen3* library.

```
using Eigen::Dynamic;
using Eigen::RowMajor;

Eigen::Matrix<double,Dynamic,Dynamic,RowMajor> A( N, N ), C( N, N );
Eigen::SparseMatrix<double,RowMajor> B( N, N );
// ... Initialization of the matrix and the vectors
C.noalias() = A * B;
```

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

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

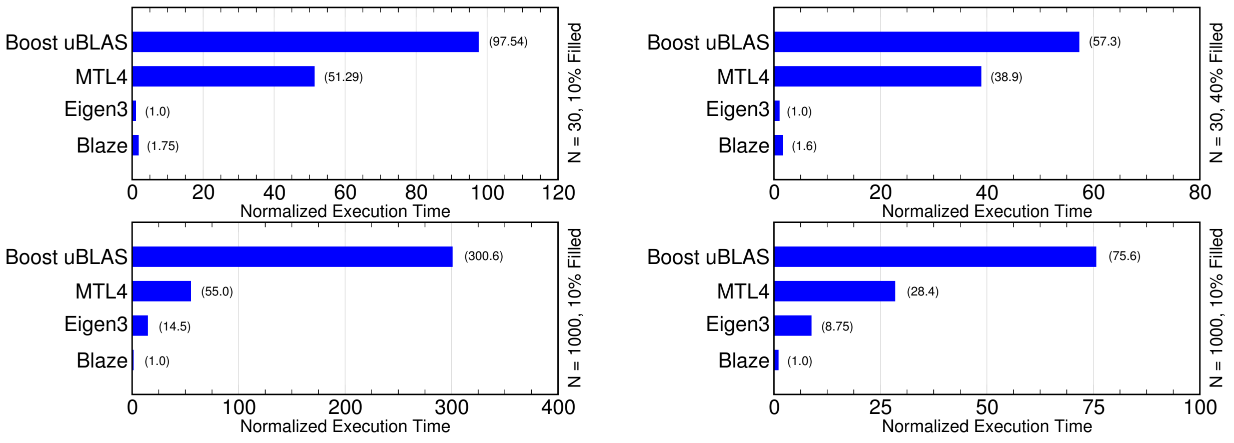

Figure 4: Performance comparison between the different ET frameworks for the multiplication of a dense matrix with a sparse matrix.

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

In some cases it is necessary for performance reasons to break the fundamental "no temporaries" rule of standard ETs. In this section we have specifically selected two examples of complex expressions that require the creation of temporaries in order to demonstrate the shortcoming of this rule.

The first case is the multiplication of a dense matrix with the sum of three dense vectors: $A \cdot (a + b + c)$. 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<double,2> A( N, N );
blitz::Array<double,1> a( N ), b( N ), c( N ), d( N );
blitz::firstIndex i;
blitz::secondIndex j;
// ... Initialization of the matrix and vectors
blitz::Array<double,1> 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<double> A( N, N );
boost::numeric::ublas::vector<double> a( N ), b( N ), c( N ), d( N );
// ... Initialization of the matrices
noalias( d ) = prod( A, ( a + b + c ) );
```

Listing 21: Use of the expression $d = A * (a + b + c)$ in the *MTL4* library.

```
mtl::dense2D<double> A( N, N );
mtl::dense_vector<double> a( N ), b( N ), c( N ), d( N );
// ... Initialization of the matrices
mtl::dense_vector<double> tmp( a + b + c );
d = A * tmp;
```

Listing 22: Use of the expression $d = A * (a + b + c)$ in the *Eigen3* library.

```
using Eigen::RowMajor;

Eigen::Matrix<double,Dynamic,Dynamic,RowMajor> A( N, N );
Eigen::Matrix<double,Dynamic,1> a( N ), b( N ), c( N ), d( N );
// ... Initialization of the matrices
d.noalias() = A * ( a + b + c );
```

Listing 23: Use of the expression $d = A * (a + b + c)$ in the *Blaze* library.

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

Listings 18 through 23 show the implementation of the complex expression with classic operator overloading, *Blitz++*, Boost *uBLAS*, *MTL4*, *Eigen3*, and *Blaze*, respectively. Interestingly it is necessary to explicitly create the temporary tmp in case of *Blitz++* and *MTL4* since it is syntactically not possible to evaluate the complex expression within a single statement. Figure 5 shows the in-cache and out-of-cache performance results of the six implementations.

For both small and large N, the *Blitz++*, Boost *uBLAS*, and *MTL4* libraries do not exhibit good performance. 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*, which mainly suffers from not

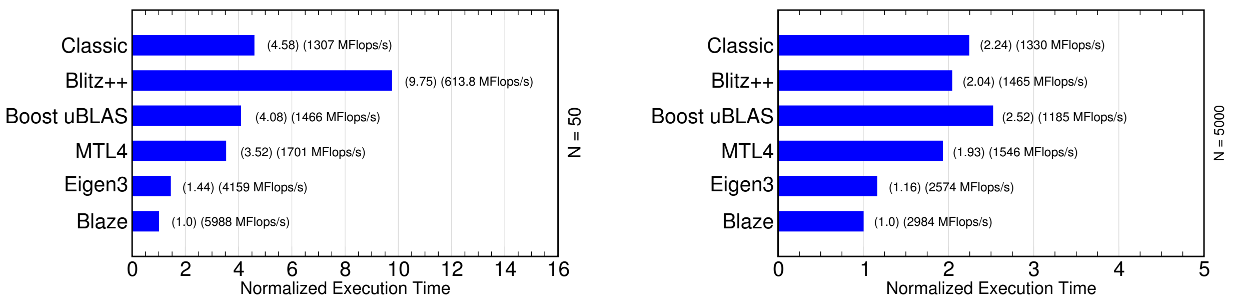

Figure 5: Performance comparison between six implementations of the complex expression $A \cdot (a + b + c)$.

evaluating the vector addition subexpression. The *Eigen3* and *Blaze* libraries, which use a single temporary to store the intermediate result of the vector additions and utilize optimized kernel functions for the subsequent matrix-vector multiplication, have a clear performance advantage. Table 2 shows hardware counter measurements, which allow a more detailed performance analysis. The repeated evaluation of the vector additions incur a $2\times$ larger number of arithmetic operations for Boost *uBLAS*. It seems surprising that *Blaze* and *Eigen3*, although they show the best performance levels, have the worst (highest) CPI value. This is because they manage to generate much fewer machine instructions, and the code is clearly bound by data transfers (as indicated by the memory bandwidth, which is close to the STREAM limit). For reference we have also included counter data for L1 data cache line replacements, i.e., how often it was necessary to refill an L1 cache line from L2, invalidating or evicting its previous contents. A large number of replacements indicates poor data locality, and shows clearly the differences in the data access patterns between the code variants.

		Memory Bandwidth [MByte/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	4259	3.33483	5.10292	0.412432	6.26337
	Blitz++	4999	2.83944	5.12711	0.405723	6.29091
	Boost *uBLAS*	3772	2.85076	10.1008	0.543944	12.5277
	MTL4	6143	2.28151	5.0069	0.420074	6.2694
	Eigen3	10337	0.65668	5.06263	0.823168	3.94277
	Blaze	10564	0.46034	5.04643	1.1158	3.94093

Table 2: Hardware counter performance analysis of the complex expression $A \cdot (a + b + c)$ for $N = 5000$.

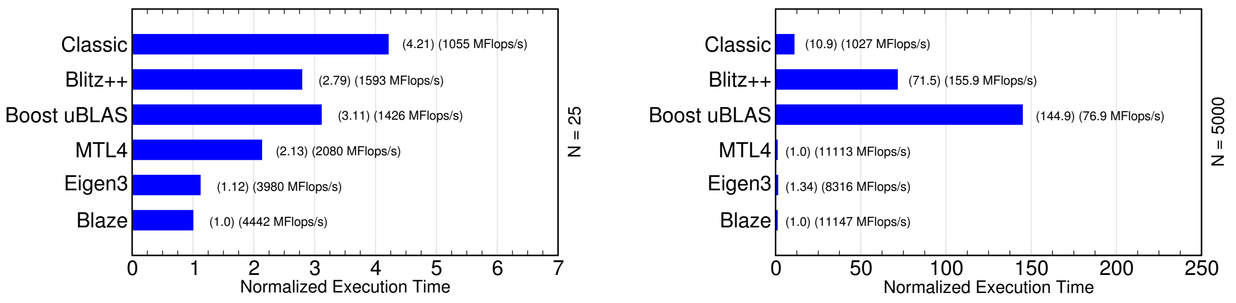

Figure 6: Performance comparison between six implementations of the complex expression $(A + B) \cdot (C - D)$.

Another interesting case of a complex expression involves four dense matrices: $E = (A+B)\cdot(C-D)$. In order to execute the matrix multiplication efficiently, both operands must be evaluated beforehand. Again, in case of *Blitz++* and *MTL4* the expression cannot be computed within a single statement, which results in two explicit temporary matrices. Again this has some benefit as can be seen from the data in Figure 6, which shows in-cache and out-of-cache performance results for classic operator overloading, *Blitz++*, Boost *uBLAS*, *MTL4*, *Eigen3*, and the *Blaze* library. *Blitz++* always performs better than Boost *uBLAS*, which does not create any temporaries and re-evaluates the matrix addition and subtraction repeatedly.

However, both *Blitz++* and Boost *uBLAS* exhibit poor performance in comparison to *MTL4*, *Eigen3*, and *Blaze*, which internally create two temporaries to hold the intermediate results of the matrix addition and subtraction and use optimized kernels to carry out the multiplication. It is particularly striking that for large N both *Blitz++* and Boost *uBLAS* are immensely outperformed by classic operator overloading, since the latter does create the necessary temporaries and, more importantly,utilizes a faster kernel. These performance values are confirmed by the hardware counter results in Table 3. Especially the large data cache replacement rates, the large CPI, and the low memory bandwidth of the "slow" frameworks indicate low code quality.

		Memory Bandwidth [MByte/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	4091	15.2473	2.50443	0.471834	31.3022
	Blitz++	628	10.2533	2.58038	4.58646	266.62
	Boost *uBLAS*	619	13.9891	6.15498	6.77575	533.426
	MTL4	578	2.05486	2.50672	0.323117	2.08345
	Eigen3	410	2.11584	2.53706	0.411949	4.05275
	Blaze	580	2.05009	2.50673	0.323406	2.07969

Table 3: Hardware counter performance analysis of the complex expression $(A + B) \cdot (C - D)$ for $N = 5000$.

Admittedly, a simple solution to improve the performance of Boost *uBLAS* would be the explicit generation of temporaries whenever necessary. However, arguably the primary goal of ETs is the ability to use infix operator notation and to provide a convenient, intuitive black box interface for all kinds of mathematical operations. Therefore a user cannot be blamed for the lack of proper automatic recognition of necessary temporaries. The standard 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 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 not to insert function calls.

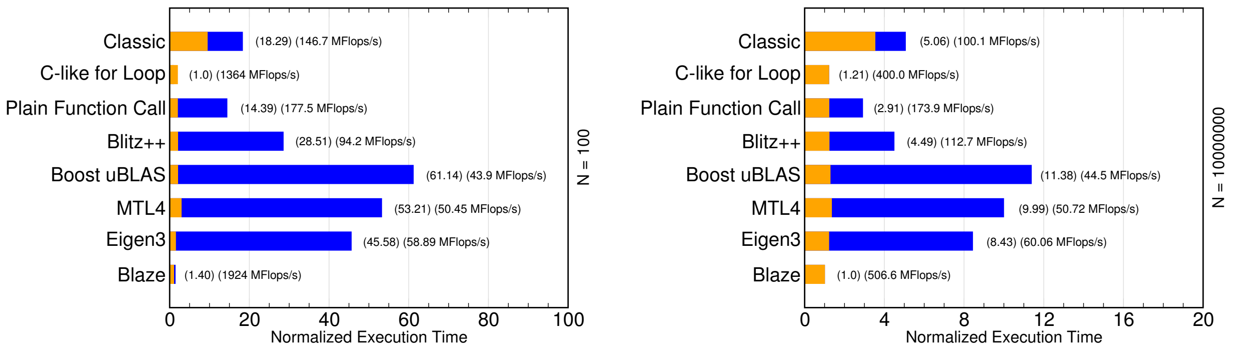

Figure 7: Performance comparison of the dense vector addition for successful and failed inlining. The short bars correspond to results from Figure 1, the long bars to the results with failed inlining. The bars are scaled according to the fastest result with successful inlining.

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 (see the compiler flags in Section 3). In order to demonstrate the impact of failed inlining, however, we switched off inlining altogether. Figure 7 shows a comparison between successful and failed inlining in case of the dense vector addition for in-cache and out-of-cache computations (the inlined performance values correspond to results from Figure 1).

As this comparison shows, inlining poses a severe and fundamental problem for all ET-based production code. This statement is also true for the ET implementation of the *Blaze* library, but due to the concept of embedding HPC-kernels *Blaze* is affected much less than the other ET frameworks. 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.

9 A New ET Methodology: Smart Expression Templates

The idea to combine high performance code with the mathematical syntax provided by the C++ operators is justified: Code clarity, readability, and maintainability are greatly improved if used in a mathematical context. However, as shown above, ETs themselves are not generally able to provide high performance. The performance of libraries using standard ETs is limited due to the abstracting nature of ETs and due to the limited optimization capabilities of compilers. Yet ET libraries are able to deliver high performance as is demonstrated by the *MTL4*, *Eigen3*, and *Blaze* libraries. Whether or not ET-based libraries are suitable for HPC is basically a matter of the underlying methodology. In this section we will introduce SETs, which conceptually make ETs compatible with the requirements of HPC.

9.1 The Methodology

In contrast to other ET approaches, SETs take a completely different approach in order to achieve the goal of combining performance and syntax. In SETs the notion of ETs being a performance optimization is completely dropped. Instead, SETs merely act as an intelligent parsing functionality that provides the following features:

1. **No abstraction from arithmetic operations and data types in order to be able to fully exploit all available knowledge.** For instance, matrix additions are treated in a fundamentally different way than matrix multiplications. This is achieved by providing the generated expression objects with the detailed knowledge about the operation and the operands. This feature is provided by *MTL4*, *Eigen3*, and *Blaze*, but not by Boost *uBLAS* and *Blitz++*.

2. **Use of architecture-specific, optimized compute kernels to achieve maximum performance.** SETs act as a wrapper around highly optimized, architecture specific compute kernels such as dgemm. Depending on the type of the operands the fastest and most specialized kernel is selected for the evaluation of the expression. In contrast to standard ETs, the evaluation of expressions is always based on optimized kernels (as already shown even in case of the dense vector addition), which minimizes the dependency on the compiler's inlining capability. Additionally, since the kernels can be easily exchanged, this kernel-based approach facilitates the transition from single- and multicore-core architectures to GPUs and massively parallel systems. Moreover, existing highly optimized low-level kernels can be easily reused and do not have to be translated to templated C++ only to arrive at a similar machine code in the end. Currently, *Blaze* is based on this philosophy, *MTL4* and *Eigen3* support it to a certain extend.

3. **Automatic selection of optimal evaluation strategies.** In addition to the selection of the compute kernel, SETs incorporate the knowledge about the optimal evaluation strategy for compound expressions. For instance, instead of evaluating the statement A = B + C $*$ D as

```
Temp = C * D;
A = B + Temp;
```

SETs evaluate the statement as

```
A = B;
A += C * D;
```

without the use of any temporary. This feature is incorporated to some extend in both the *Eigen3* and *Blaze* library.

4. **Ability to restructure the expression to optimize the evaluation.** 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 is evaluated first, the performance can be dramatically improved since the matrix-matrix multiplication can be avoided in favor of a second matrix-vector multiplication. Figure 8 gives an example for the two ET libraries that syntactically allow to formulate the expression within a single statement:

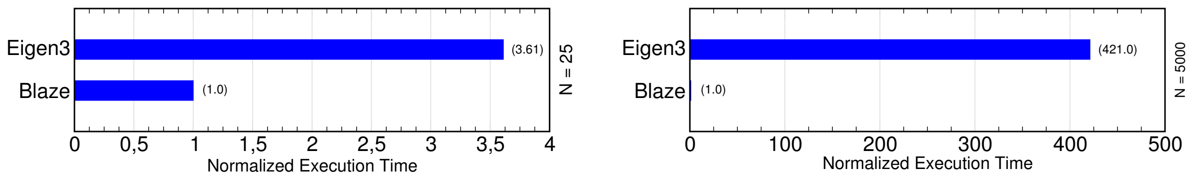

Figure 8: Performance comparison of the complex expression $A \cdot B \cdot v$.

In contrast to *Eigen3*, *Blaze* restructures the expressions based on performance considerations. Considering the tremendous performance difference, it is apparent that in case a numerics library syntactically allows such expressions, the implementation must be aware of the possibility to improve performance by restructuring them. Currently, only *Eigen3* and *Blaze* support this feature to a certain level.

5. **Ability to specific, operand-based creation of necessary temporaries.** As shown in Section 7, the creation of temporaries can be the key to high performance in case the syntax of the ET implementation allows the formulation of complex expressions. In case of the standard ET methodology, due to the abstraction the advantage of creating a temporary cannot be recognized. SETs recognize the need for an intermediate evaluation of operands based on the knowledge of the arithmetic operation and the operands. For instance, in case of the complex expression $(A{+}B){\cdot}(C{-}D)$, it is beneficial to first evaluate the left-hand side and right-hand side operands prior to the matrix multiplication such that subsequently the dgemm kernel can be used. In their current implementation, *MTL4*, *Eigen3*, and *Blaze* support this feature, Boost *uBLAS*and *Blitz++*do not.

6. **Automatic detection of aliasing effects.** In the current ET libraries, aliasing effects such as in the statement $\mathrm{x} = \mathrm{A} * \mathrm{x}$; are either not handled at all (*Blitz++*), result in runtime errors (*MTL4*) or must be explicitly handled (Boost *uBLAS*, *Eigen3*). SETs are able to detect aliasing effects automatically and can therefore introduce the necessary temporaries automatically and efficiently. Currently only *Blaze* supports this feature.

9.2 The Smart Expression Template Implementation of *Blaze*

In the following we will sketch the implementation of the six key ingredients of SETs in the *Blaze* library by means of the expression $A \cdot B \cdot v$. Note however that the *Blaze* implementation is only one possible implementation of SETs and therefore merely serves as an illustrating example to SETs. Additionally note, that we have slightly simplified the source code to facilitate code presentation and comprehension. For instance, we omit all functionality that differentiates between row-wise and column-wise stored matrices and transposed and non-transposed vectors.

Listing 24 shows the implementation of the expression by means of the *Blaze* functionality. Since the right-hand side expression is evaluated from left to right, the first operation to be evaluated is the leading matrix-matrix multiplication. Listing 25 shows the implementation of the operator that after an initial check of the dimensions of the matrices only returns a temporary expression object of type DMatDMatMultExpr, which represents the multiplication of two dense matrices.

Listing 24: Use of the expression $w = A \cdot B \cdot v)$ in the *Blaze* library.

```
blaze::MatrixMxN<double> A, B;
blaze::Vector v, w;
// Initialization of the matrices and vectors
w = A * B * v;
```

Listing 25: Multiplication operator for the matrix-matrix multiplication

```
template< typename T1    // Type of the left-hand side dense matrix
        , typename T2 >  // Type of the right-hand side dense matrix
inline const DMatDMatMultExpr<T1,T2>
   operator*( const DenseMatrix<T1>& lhs, const DenseMatrix<T2>& rhs )
{
   if( (~lhs).columns() != (~rhs).rows() )
      throw std::invalid_argument( "Matrix sizes do not match" );

   return DMatDMatMultExpr<T1,T2>( ~lhs, ~rhs );
}
```

Listing 26 shows part of the implementation of the according class template. The template parameter MT1 represents the type of the left-hand side dense matrix operand, the parameter MT2 represents the type of the right-hand side dense matrix operand. The DMatDMatMultExpr class derives both from the Expression class (which formally tags it as an expression) and from the DenseMatrix class template (using the "curiously recurring template pattern (CRTP) [15, 24]), which makes it a dense matrix. The type safe up-cast that is enabled by CRTP is provided via **operator~** (see for instance Listing 24, line 6 and 9). Via common "Template Meta Programming" (TMP) [1] techniques the data types of the two operands are used to evaluate the two member data types Lhs and Rhs. The two operands of the matrix multiplication are either stored by value in case they are expression objects (i.e. derived from the class Expression) or by reference in case they are a non-expression, plain dense matrix type. Due to that expression objects are always light-weight objects that are cheap to copy.

Listing 26: 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 : public DenseMatrix< DMatDMatMultExpr<MT1,MT2> >
                       , private Expression
{
 public:
  // Public interface omitted

 private:
  // ...

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

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

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

  template< typename VT >  // Type of the right-hand side dense vector
  friend inline const DMatDVecMultExpr< MT1, DMatDVecMultExpr<MT2,VT> >
    operator*( const DMatDMatMultExpr& lhs, const DenseVector<VT>& rhs )
  {
    typedef DMatDVecMultExpr<MT2,VT>         InnerType;
    typedef DMatDVecMultExpr<MT1,InnerType>  OuterType;
    return OuterType( lhs.lhs_, InnerType( lhs.rhs_, ~rhs ) );
  }

  // ...
};
```

The next operation to be evaluated is the matrix-vector multiplication of the matrix-matrix multiplication $A{\cdot}B$ with the vector v. In order to split the matrix-matrix multiplication to generate two matrix-vector multiplications a special multiplication operator is required. This multiplication operator is defined as friend function inside the DMatDMatMultExpr class body. Via the Barton-Nackman trick [17, 24] the operator is injected into the surrounding namespace. In case the just created matrix multiplication object is multiplied with any dense vector object, this operator is used and the matrix-matrix multiplication is restructured into two nested matrix-vector multiplications (i.e. two DMatDVecMultExpr objects).

Listing 27 shows part of the implementation of the DMatDVecMultExpr class template that represents multiplications of a dense matrix and a dense vector. The template parameter MT represents the type of the left-hand side dense matrix operand, the parameter VT represents the type of the right-hand side dense vector operand. DMatDVecMultExpr is also derived from the Expression class as well as from the DenseVector class template. Again, Lhs and Rhs are the member data types for the two operands.

Listing 27: Smart expression object for the matrix-vector multiplication (I)

```
template< typename MT    // Type of the left-hand side dense matrix
        , typename VT >  // Type of the right-hand side dense vector
class DMatDVecMultExpr : public DenseVector< DMatDVecMultExpr<MT,VT> >
                       , private Expression
{
 public:
  // Public interface omitted
  // ...

  typedef typename MT::ResultType  MRT;  // Result type of the matrix expression.
  typedef typename VT::ResultType  VRT;  // Result type of the vector expression.

  // Result type for expression template evaluations.
  typedef typename MathTrait<MRT,VRT>::MultType  ResultType;

  // Resulting element type.
  typedef typename ResultType::ElementType  ElementType;

  // Data type for composite expression templates.
  typedef const ResultType  CompositeType;

 private:
  // ...

  // Composite type of the left-hand side matrix expression.
  typedef typename SelectType< IsExpression<MT>::value, const MT, const MT& >::Type Lhs;

  // Composite type of the right-hand side vector expression.
  typedef typename SelectType< IsExpression<VT>::value, const VT, const VT& >::Type Rhs;

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

  // ...
};
```

After the evaluation of the two multiplication operators, the restructured expression is assigned to the left-hand side dense vector. For this purpose, the VectorN class template implements a special assignment operator for the assignment of DenseVectors (see Listing 28).

Listing 28: Expression template assignment operator of the VectorN class

```
1  template< typename Type >  // Data type of the vector
2  template< typename VT >    // Type of the right-hand side dense vector
3  inline VectorN<Type,TF>& VectorN<Type,TF>::operator=( const DenseVector<VT>& rhs )
4  {
5     using blaze::assign;
6
7     if( IsExpression<VT>::value && (~rhs).isAliased( this ) ) {
8        VectorN tmp( rhs );
9        swap( tmp );
10    }
11    else {
12       resize( (~rhs).size() );
13       assign( *this, ~rhs );
14    }
15
16    return *this;
17 }
```

During the assignment, two possible scenarios have to be distinguished: In case the left-hand side target vector is aliased with the expression on the right-hand side (see line 7), a temporary vector is created and the “temporary swap” idiom [10, 9] is applied. In case no aliasing can be detected, the vector is resized accordingly and assigned the right-hand side expression. The call to the assign function in line 13 is the core of the evaluation of the expression. Each expression object can define its own assign kernel via namespace injection. Listing 29 shows the implementation of the according assign function of the DMatDVecMultExpr class template along with the necessary type definitions.

Listing 29: Smart expression object for the matrix-vector multiplication (II)

```
 1 template< typename MT    // Type of the left-hand side dense matrix
 2         , typename VT >  // Type of the right-hand side dense vector
 3 class DMatDVecMultExpr : private Expression
 4 {
 5  public:
 6   // Public interface omitted
 7
 8  private:
 9   // ...
10
11   typedef typename MRT::ElementType   MET;  // Element type of the matrix expression.
12   typedef typename VRT::ElementType   VET;  // Element type of the vector expression.
13   typedef typename MT::CompositeType  MCT;  // Composite type of the matrix expression.
14   typedef typename VT::CompositeType  VCT;  // Composite type of the vector expression.
15
16   // Compilation switch for the composite type of the left-hand side matrix expression
17   enum { blas = IsExpression<MT>::value &&
18                 ( IsFloat<MET>::value || IsDouble<MET>::value ) &&
19                 ( IsFloat<VET>::value || IsDouble<VET>::value ) };
20
21   // Type for the assignment of the left-hand side matrix operand.
22   typedef typename SelectType< blas, const MRT, MCT >::Type  LT;
23
24   // Type for the assignment of the the right-hand side matrix operand.
25   typedef typename SelectType< IsExpression<VT>::value, const VRT, VCT >::Type  RT;
26
27   // Specialized assign function injected into the surrounding namespace
28   template< typename VT2 >  // Type of the target dense vector
29   friend inline void assign( DenseVector<VT2,false>& lhs,
30                              const DMatDVecMultExpr& rhs )
31   {
32     LT A( rhs.mat_ );  // Evaluation of the left-hand side dense matrix operand
33     RT x( rhs.vec_ );  // Evaluation of the right-hand side dense vector operand
34
35     DMatDVecMultExpr::selectAssignKernel( ~lhs, A, x );
36   }
37
38   // ...
39 };
```

The first step of the evaluation of a matrix-vector multiplication within the assign function is the evaluation of the two operands. LT corresponds to the data type for the evaluation of the left-hand side dense matrix operand. In case the matrix operand is an expression and the element types of both the matrix as well as the vector operand would enable the application of the dgemv BLAS function, an intermediate evaluation is enforced by using the resulting data type ResultType of the matrix operand. Else the nested CompositeType of the matrix expression is used, which incorporates the knowledge of the expression itself whether an intermediate evaluation is required or not. In case an evaluation is necessary, LT corresponds to a non-reference, concrete data type, otherwise LT is a mere reference to the operand. Via a similar procedure the data type for the evaluation of the right-hand side dense vector operand is determined. However, in this case it is enough to differentiate whether VT is an expression type (i.e. derived from the Expression class) or a non-expression, plain vector type.

In the example of the two nested matrix-vector multiplications in the expression $A \cdot B \cdot v$, in case of the inner matrix-vector multiplication $B \cdot v$, both operands do not require any intermediate evaluations. Therefore both LT and RT correspond to references to the original operands. In case of the outer matrix-vector multiplication, the vector operand corresponds to the inner matrix-vector multiplication and therefore requires an intermediate evaluation prior to the execution of the outer matrix-vector multiplication. Whereas for LT a reference to the matrix operand is used, RT corresponds to the most suitable result type for a matrix-vector multiplication of a MatrixMxN<**double**> with a VectorN<**double**>, which is determined via the MathTrait class template (see Listing 27, line 14). For this example, a temporary VectorN<**double**> is used, which is constructed in place via the same assign mechanism.

In line 35, the evaluated operands are passed to the selectAssignKernel function (see Listing 30). Via the "substitution failure is not an error" (SFINAE) principle (see [24]) exactly one kernel is available depending on the data types of the evaluated operands. In case the two operands as well as the target vector are compatible to the BLAS standard, the dgemv kernel is selected, else a slower, but more generally applicable default kernel is used.

Listing 30: Assign functions for the evaluation of the matrix-vector multiplication

```
template< typename MT    // Type of the left-hand side dense matrix
        , typename VT >  // Type of the right-hand side dense vector
class DMatDVecMultExpr : private Expression
{
 private:
  // ...

  template< typename VT1    // Type of the left-hand side target vector
          , typename MT1    // Type of the left-hand side matrix operand
          , typename VT2 >  // Type of the right-hand side vector operand
  static inline typename boost::enable_if_c< !IsBlasCompatible<VT1>::value ||
                                             !IsBlasCompatible<MT1>::value ||
                                             !IsBlasCompatible<VT2>::value >::Type
    selectAssignKernel( VT1& y, const MT1& A, const VT2& x )
  {
    // Default implementation of the matrix-vector multiplication
    // ...
  }

  template< typename VT1    // Type of the left-hand side target vector
          , typename MT1    // Type of the left-hand side matrix operand
          , typename VT2 >  // Type of the right-hand side vector operand
  static inline typename boost::enable_if_c< IsBlasCompatible<VT1>::value &&
                                             IsBlasCompatible<MT1>::value &&
                                             IsBlasCompatible<VT2>::value >::Type
    selectAssignKernel( VT1& y, const MT1& A, const VT2& x )
  {
    // Utilization of the cblas_dgemv kernel
    // ...
  }

  // ...
};
```

In this implementation of the evaluation of the expression $A \cdot B \cdot v$ a total of eight function calls is used, among these two calls to the BLAS dgemv function, and a single temporary is automatically created for the inner matrix-vector multiplication.

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 substitute?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.

References

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

[2] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Ei-jkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1993.

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

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

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

[6] Boost Lambda library. Homepage of the Boost Lambda library: http://www.boost.org/doc/libs/1_-45_0/doc/html/lambda.html.

[7] Boost uBLAS library. Homepage of the Boost uBLAS library: http://www.boost.org/doc/libs/1_-45_0/libs/numeric/ublas/doc/index.htm.

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

[9] H. Sutter. *More Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 2007.

[10] H. Sutter. *Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 2008.

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

[12] 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.

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

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

[15] J. Coplien. Curiously Recurring Template Patterns. *C++ Report*, 7(2):24–27, 1995.

[16] 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.

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

[18] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, VA, 1991-2007. A continually updated technical report.

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

[20] MTL4 library. Homepage of the MTL4 library: http://www.simunova.com/de/mtl4.

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

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

[23] 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.

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