High Performance Smart Expression Template Math Libraries

Klaus Iglberger*, Georg Hager†, Jan Treibig† and Ulrich Rüde‡
*Central Institute for Scientific Computing, University Erlangen-Nuremberg, Erlangen, Germany
Email: klaus.iglberger@zisc.uni-erlangen.de
†Erlangen Regional Computing Center, University Erlangen-Nuremberg, Erlangen, Germany
Email: {georg.hager,jan.treibig}@rrze.fau.de
‡Chair for System Simulation, University Erlangen-Nuremberg, Erlangen, Germany
Email: ulrich.ruede@informatik.uni-erlangen.de

***Abstract*—Performance is of utmost importance for linear algebra libraries since they usually are the core of numerical and simulation packages and use most of the available compute time and resources. However, especially in large scale simulation frameworks the readability and ease of use of mathematical expressions is essential for a continuous maintenance, modification, and extension of the software framework. Based on these requirements, in the last decade C++ Expression Templates have gained a reputation as a suitable means to combine an elegant, domain-specific, and intuitive user interface with "HPC-grade" performance. Unfortunately, many of the available ET-based frameworks fall short of the expectation to deliver high performance, adding to the general mistrust towards C++ math libraries. In this paper we present performance results for Smart Expression Template libraries, demonstrating that by proper combination of high-level C++ code and low-level compute kernels both requirements, an elegant interface and high performance, can be achieved.**

I. Motivation

Math libraries are an essential component of most numerical packages and simulation codes. A primary reason for that is that the application of existing and well tested software saves considerable time and effort. More importantly, programmers rely on the efficient implementation of the underlying math kernels. Therefore the requirements on the performance of math libraries are extremely high. However, next to performance, the readability of mathematical expressions and the ease of use of the math kernels are two other important prerequisites.

The following example demonstrates a call to the C/C++ version of the `dgemm` function from the BLAS standard, which performs a multiplication of two dense matrices of size $N \times N$:

Listing 1: Matrix multiplication via a call to the `cblas_dgemm` function.

```
1 double* A, *B, *C;
2 // ... Initialization of the matrices
3 cblas_dgemm( CblasRowMajor,
4               CblasNoTrans,
5               CblasNoTrans,
6               N, N, N, 1.0, A, N, B, N, 0.0, C, N );
```

On the positive side, users of this function can count on a very efficient implementation, and therefore on very high performance, if they employ an optimized BLAS library. The disadvantage, however, is the use of the function: Due to the extremely high number of non-self-explanatory function parameters the readability of the code is severely limited. Additionally, the function call strongly depends on correct matrix properties: changes to the matrix size, padding, etc. also require changes to the function arguments. This dependency increases the rigidity and fragility of the code and can lead to errors that are hard to detect and thus may take a long time to track down.

A much better solution in terms of readability and ease of use, and therefore also programmability, maintainability, and extensibility is the natural, elegant, and intuitive syntax of domain-specific languages. The C++ approach to this problem is operator overloading, which provides a syntax as close as possible to the corresponding mathematical expression:

Listing 2: C++-style matrix multiplication.

```
1 Matrix<double> A( N, N ), B( N, N ), C( N, N );
2 // ... Initialization of the matrices
3 C = A * B;
```

Any changes to the matrix sizes, including padding, are automatically taken into account and therefore do not lead to changes in the expression, which improves the robustness of the code. However, classic C++ operator overloading involves additional overhead due to the creation of temporary objects. A possible solution are Expression Templates [1] (ET), which perform lazy evaluation of the mathematical expression to avoid any unnecessary overhead. They also promise to combine high performance with elegant syntax. The performance-improving characteristics of ETs in comparison to classic operator overloading, however, are mainly limited to BLAS level 1 operations. In fact, as described in [2], in contrast to common lore, ETs are not an optimization technique but an abstraction technique, which raises false expectations: The combination of over-confidence in compiler optimization and aggressive inlining leads to bad performance in many C++ ET-based libraries. This fact adds to the general mistrust against C++-based math libraries. However, in order to design robust and reliable software and to satisfy both the requirements on performance and syntax, it is of highest importance to combine good performance with a good interface.

978-1-4673-2362-8/12/$31.00 ©2012 IEEE

A solution as described in [2] is the Smart Expression Template (SET) method. In contrast to standard ETs, SETs drop the notion of ETs being a performance optimization. Instead, SETs merely act as an intelligent wrapping structure around highly optimized math kernels. Whereas standard ETs rely on automatic low-level compiler optimization, SET-based libraries provide explicit optimization by either manually vectorized compute kernels or the internal use of optimized libraries like the MKL. Additionally, SETs have the ability to restructure expressions in order to optimize their evaluation and regain the ability to create specific, operand-based temporaries whenever beneficial for the performance.

The paper is organized as follows. In Section II we give a short overview of the used (S)ET math libraries, before Section III briefly summarizes the details of our benchmark platform. In Section IV we will compare the performance results of BLAS level 1, 2, and 3 operations with dense matrices and vectors of eight (S)ET-based math libraries with classic C++ operator overloading as well as the Intel MKL. Additionally we will analyze the application performance of these libraries with an implementations of the conjugate gradient (CG) algorithm. Thereby we will demonstrate the general suitability of the SET methodology for HPC in terms of performance and the advantages in comparison to plain BLAS function calls in terms of software development. Section V concludes the paper and provides suggestions for future work.

II. Related Work

The first ET-based C++ library for dense arithmetic was *Blitz++* [3]. This framework, written by the inventor of ETs, Todd Veldhuizen, has been recognized as a pioneer in the area of C++ template metaprogramming [4]. The Boost *uBLAS* library [5] is one of the most widespread ET math libraries since it is distributed together with the Boost library [6]. In contrast to *Blitz++* it additionally provides sparse matrices and vectors. The *Armadillo* library [7] is restricted to dense linear algebra operations, but employs SETs to integrate BLAS and LAPACK for optimized performance. The same feature is provided by *MTL4*, which additionally includes sparse matrix operations. An alternative with similar functionality is the *GMM++* library [8], which currently only supports ATLAS [9]. Numerics involving dense and sparse matrices and vectors, the use of optimized kernels, and the advanced SET features of intrinsics-based vectorization of non-BLAS operations and automatic expression optimization are supported by the *Eigen3* library [10] and the *Blaze* library [11]. In contrast to *Eigen3*, which provides optimized kernels for all basic operations, *Blaze* uses BLAS subroutines for BLAS level 2 and 3 operations.

In [2] we have analyzed several of these ET implementations in detail and have introduced the notion of SETs and our SET library *Blaze* in particular. In this work we continue our analysis and expand it to more ET-based libraries, focus on the optimization and vectorization capabilities for dense arithmetic, and present performance results for the CG algorithm, which is fundamental for many applications.

III. Benchmark Platform

A quad-core Intel Xeon CPU E3-1280 (“Sandy Bridge”) CPU at 3.5 GHz base frequency with 8 MByte of shared L3 cache was used for all benchmarks. The maximum achievable memory bandwidth (as measured by the STREAM benchmark [12]) is about 18.5 GByte/s. In contrast to other x86 processors, this limit can be hit by a single thread if the code is strongly memory bound. Each core has a theoretical peak performance of eight flops per cycle in double precision (DP) using AVX vector instructions. AVX (“Advanced Vector Extensions”) is Intel’s variant of “Single Instruction Multiple Data” (SIMD) parallelism on their latest processor designs. A core can execute one AVX add and one AVX multiply operation per cycle. Full in-cache performance can only be achieved with SIMD-vectorized code. This includes loads and stores, which exist in full-width (AVX) vectorized, half-width (SSE) vectorized, and “scalar” variants. A maximum of one 256-bit wide AVX load and one 128-bit wide store can be sustained per cycle. 256-bit wide AVX stores thus have a two-cycle throughput.

Due to the “Turbo Mode” feature the processor can increase the clock speed by up to 400 MHz, depending on load and temperature. Since we use a single core only in our benchmarks, the CPU ran continuously at 3.9 GHz. We will use this frequency as an input parameter to the L1 performance models (see Sect. IV).

For benchmarking we run the according kernels many times:

Listing 3: Structure of the benchmarks

```
// Start of the time measurements
for( int iter=0; iter<NITER; ++iter ) {
   // ... Execution of the benchmark kernel
}
// End of the time measurements
```

`NITER` is chosen such that the overall runtime is large enough to be measured accurately. This has the effect that for small vectors and matrices, for which the data structures fit into the caches, the benchmark probes the capability of the cache levels and/or compute units. At larger loop lengths the runtime is dominated by main memory access.

The GNU g++ 4.6.1 compiler was used with the following compiler flags:

Listing 4: GCC compilation flags

```
g++ -Wall -Wshadow -Woverloaded-virtual -ansi
    -pedantic -O3 -mavx -DNDEBUG
```

We use the latest *Blitz++* implementation (available via anonymous CVS access), Boost *uBLAS* in version 1.46, GMM++ in version 4.1, Armadillo in version 2.4.2, *MTL4* in version 4.0.8368, *Eigen3* in version 3.1.0alpha1, and the Intel MKL in version 10.3 (update 8). All libraries were benchmarked as given. We only present double precision results in MFlop/s graphs for each test case. For all in-cache benchmarks we make sure that the data has already been loaded to the cache.

IV. Performance Results

A. BLAS level 1 operations

Figure 1 shows the daxpy performance results of all (S)ET-based math libraries, classic C++ operator overloading, and the Intel MKL for vector sizes ranging from 1 to 10000000. For large vector sizes the performance is given by the achievable main memory bandwidth of the processor. Since two 8-byte loads and one store (24 bytes in total) have to be sustained per iteration (2 flops), we expect (18.5 GByte/s)/(12 byte/flop)=1.54 GFlop/s, which is almost exactly met by all variants except classic operator overloading. At small vector sizes (fitting into the L1 cache) the performance is limited by load/store throughput; due to the sustained load/store capability of the core we expect the daxpy loop to run at half-peak performance (15.6 GFlop/s), of which 90% are reached by *Blaze* and MKL. The strong sawtooth pattern visible in the *Blaze* result in the range between 10 and 100 is caused by an explicit four-way loop unrolling on top of the SIMD vectorization, leading to a scalar (non-vectorized) remainder loop. MKL does not show this effect due to massive multi-versioning (i.e., remainder loops are unrolled and vectorized to the possible limit). However, the library call overhead makes it quite slow for small loop lengths, where *Blaze* is efficient because it generates fully inlined code.

Predicting the performance of loop kernels with data coming from outer-level (L2/L3) caches is out of scope for this work and requires more insight into the architecture and specific properties of the memory hierarchy [13].

In this performance comparison the classic C++ operator overloading shows by far the worst performance due to the extra data transfer caused by the temporary vector. It becomes apparent that the overhead due to the creation of a temporary prevents good performance. In comparison, ET-based libraries such as Boost *uBLAS*, *MTL4*, *GMM++*, *Armadillo*, and *Blitz++* offer better performance due to avoiding the temporary result. However, only *Blaze* can compete with the performance of the Intel MKL due to an explicit use of AVX SIMD intrinsics.

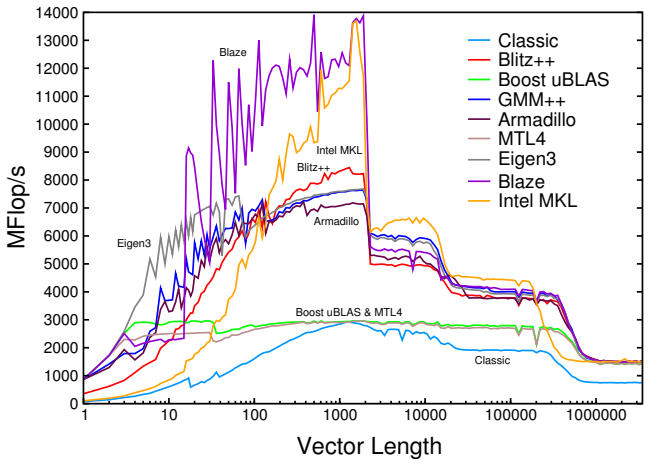

Figure 1: Performance of the daxpy product ($b \mathrel{+}= a * s$)

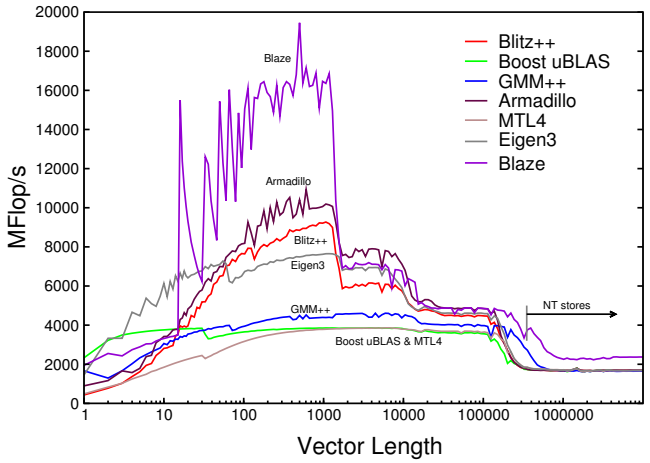

Figure 2: Performance of scaled dense vector addition ($c = s1 * a + s2 * b$)

Listing 5: Daxpy product via a call to the `cblas_daxpy` function.

```
double* a, *b;
// ... Initialization of the vectors
cblas_daxpy( N, 0.001, a, 1, b, 1 );
```

Listing 6: Daxpy product in the *Blaze* library.

```
blaze::DynamicVector<double> a( N ), b( N );
// ... Initialization of the vectors
b += a * 0.001;
```

Listing 5 and Listing 6 show the source code for the daxpy product for the direct use of the Intel MKL and the *Blaze* library, respectively. Considering that both approaches result in very similar performance, the readability, programmability, and maintainability of the *Blaze* SET library is highly superior and therefore the preferred way for creating robust, reliable C++ code.

Figure 2 shows the result of a scaled dense vector addition, for which no BLAS kernel is available. In this case, performance depends strongly on the vectorization capability of the (S)ET libraries. Within the L1 cache the load/store-based limit for this operation is the same as the arithmetic limit, namely 3/4 of peak (23.4 GFlop/s), of which *Blaze* achieves about 85%. In memory, the use of non-temporal stores on the store stream can avoid the write-allocate transfers, boosting the theoretical maximum from 1.73 GFlop/s to 2.3 GFlop/s. *Blaze* does employ non-temporal stores for large vector lengths and is thus about 33% faster than the competition, as predicted by the model.

In case no optimized BLAS kernel is available and if the expression can be efficiently evaluated element-wise (as in case of BLAS level 1 operations), *Eigen3* and *Blaze* use the standard ET methodology to allow the compiler to assemble vectorized kernels via inlining. Listing 7 gives an impression of the vectorized kernel (using SIMD intrinsics supported by the compiler) of the *Blaze* library after a complete inlining of

all *Blaze* function calls.

Listing 7: SET double-precision kernel of the *Blaze* implementation.

```
double* a = ...;  // 16-byte aligned vector array
double* b = ...;  // 16-byte aligned vector array
double* c = ...;  // 16-byte aligned vector array

const double scalar1 = ...;
const double scalar2 = ...;

const __m128d s1( _mm_set1_pd( scalar1 ) );
const __m128d s2( _mm_set1_pd( scalar2 ) );

if( size > 350000UL ) // Cache-size depending value
{
  const size_t end( size - size % 2 );

  for( size_t i=0UL; i<end; i+=2 ) {
   __m128d x = _mm_load_pd( a+i );
   __m128d y = _mm_load_pd( b+i );
   __m128d z = _mm_add_pd( _mm_mul_pd( x, s1 ),
                           _mm_mul_pd( y, s2 ) );
   _mm_stream_pd( c+i, z );  // NT stores
  }
  for( size_t i=end; i<size; ++i ) {
   c[i] = a[i] + b[i];
  }
}
else
{
  const size_t end( size - size % 8UL );

  for( size_t i=0UL; i<end; i+=8UL ) {
   __m128d x1 = _mm_load_pd( a+i );
   __m128d y1 = _mm_load_pd( b+i );
   __m128d z1 = _mm_add_pd( _mm_mul_pd( x1, s1 ),
                            _mm_mul_pd( y1, s2 ) );
   _mm_store_pd( c+i, z1 );

   __m128d x2 = _mm_load_pd( a+i+2UL );
   __m128d y2 = _mm_load_pd( b+i+2UL );
   __m128d z2 = _mm_add_pd( _mm_mul_pd( x2, s1 ),
                            _mm_mul_pd( y2, s2 ) );
   _mm_store_pd( c+i+2UL, z2 );

   __m128d x3 = _mm_load_pd( a+i+4UL );
   __m128d y3 = _mm_load_pd( b+i+4UL );
   __m128d z3 = _mm_add_pd( _mm_mul_pd( x3, s1 ),
                            _mm_mul_pd( y3, s2 ) );
   _mm_store_pd( c+i+4UL, z3 );

   __m128d x4 = _mm_load_pd( a+i+6UL );
   __m128d y4 = _mm_load_pd( b+i+6UL );
   __m128d z4 = _mm_add_pd( _mm_mul_pd( x4, s1 ),
                            _mm_mul_pd( y4, s2 ) );
   _mm_store_pd( c+i+6UL, z4 );
  }
  for( size_t i=end; i<size; ++i ) {
   c[i] = a[i] + b[i];
  }
}
```

B. BLAS level 2 operations

Figure 3 shows the performance results for classic C++ operator overloading, *Blitz++*, *uBLAS*, *MTL4*, *Eigen3*, *Blaze*, and the Intel MKL for the multiplication between a row-major dense matrix and a dense vector. The *GMM++* and *Armadillo* libraries are missing in this benchmark since they only support column-major matrices. This operation can theoretically reach half-peak performance for data sets that fit into the L1 cache. For large problem sizes, assuming that the right-hand side vector resides in cache, the performance is limited by the memory transfers for the matrix to (18.5 GByte/s)/(8 byte/2 flops)≈4.63 GFlop/s.

Apparently there exist two different performance levels for this operation: *uBLAS*, *Blitz++*, and *MTL4* perform similar to classic operator overloading, whereas *Eigen3* and *Blaze* are close to the optimized MKL. *Blaze* uses its own ET-based MVM kernel for very small problem sizes and calls MKL for $N \geq 12$. Due to inlining and vanishing function call overhead, *Blaze* and *Eigen3* are slightly faster than MKL for small vectors.

This result demonstrates the methodology advantage of SET libraries against standard ET libraries by using optimized compute kernels. In this particular case, *Blaze* internally uses the Intel MKL, and *Eigen3* provides its own implementation. Standard ET-based libraries, however, do not live up to the expectation to provide faster matrix/vector multiplications than classic C++ operator overloading.

Listing 8: Row-major matrix/vector multiplication via a call to the `cblas_dgemv` function.

```
double* A, *a, *b;
// ... Initialization of the matrix and vectors
cblas_gemv( CblasRowMajor, CblasNoTrans,
            N, N, 1.0, A, N, a, 1, 0.0, b, 1 );
```

Listing 9: Row-major matrix/vector multiplication in the *Blaze* library.

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

Listings 8 and 9 again demonstrate the advantages of the SET domain-specific syntax in comparison to a direct call to the `cblas_dgemv` function. Changes to the size of the matrix and/or vectors or changing the storage order of the matrix do not require modifications in the call to the compute kernel.

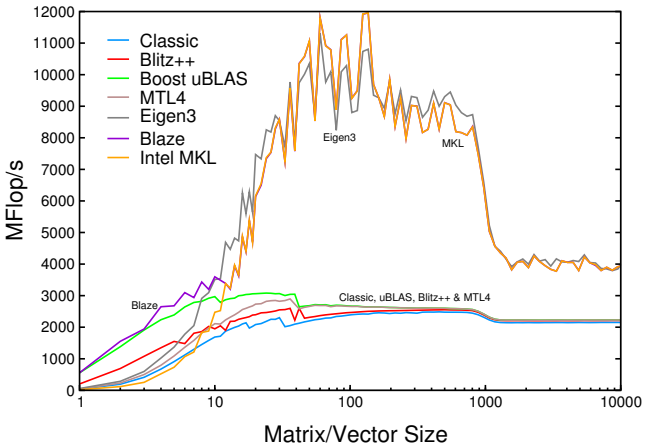

Figure 3: Performance of row-major matrix/vector multiplication ($b = A * a$)

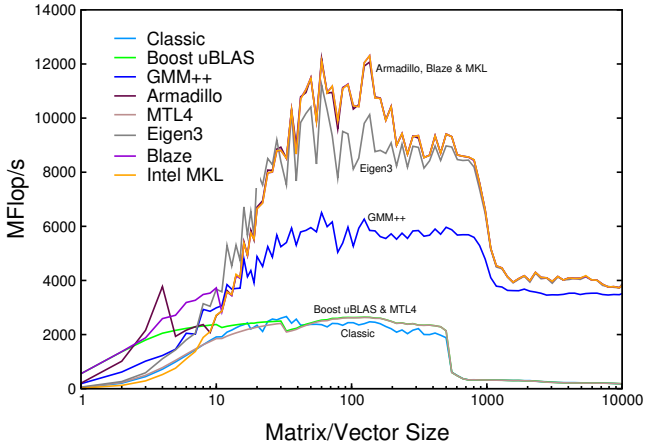

Figure 4: Performance of column-major matrix/vector multiplication ($b = A^T * a$)

Additionally, the ease of use is strongly improved without the loss of flexibility provided by the large number of function parameters of the `cblas_dgemv` function (i.e., it is for instance intuitively possible to add scalar factors).

In Figure 4 the results for the multiplication between a column-major dense matrix and a dense vector are summarized. The *Blitz++* library is missing in this benchmark because of incorrect calculation results. Whereas *uBLAS* and *MTL4* again exhibit no better performance than classic operator overloading for large vectors, *GMM++* is close to and *Armadillo*, *Eigen3*, and *Blaze* have similar performance to the Intel MKL. For this benchmark, both *Armadillo* and *Blaze* internally use the MKL kernel, whereas *Eigen3* provides its own optimized kernel. It is especially noteworthy that *Eigen3*, *Blaze*, and the Intel MKL achieve the same performance level for the column-major matrix/vector multiplication as for the row-major matrix/vector multiplication, which is an indicator for proper cache optimization.

C. BLAS level 3 operations

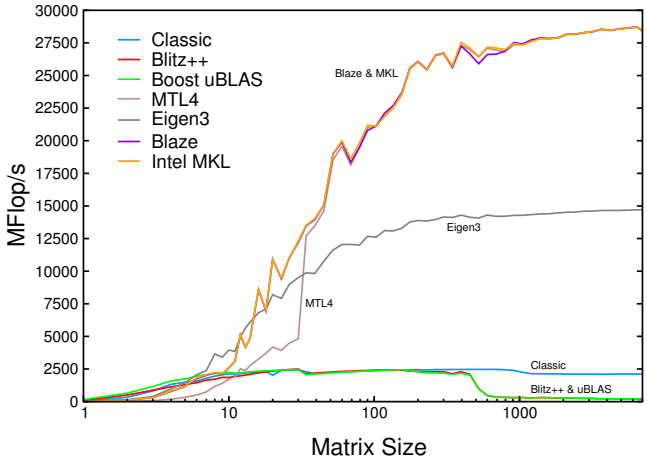

Figure 5: Performance of row-major matrix multiplication ($C = A * B$)

Due to the beneficial proportion between arithmetic operations and data transfers, it is possible to achieve peak performance for the matrix/matrix multiplication. In Figure 5 we summarize the results for a row-major matrix/matrix multiplication. *MTL4*, *Eigen3*, *Blaze*, and the MKL reach near-peak performance for large matrices. Both the *MTL4* and *Blaze* internally use the MKL, but *Eigen3* provides its own kernel. In contrast, *Blitz++* and Boost *uBLAS* even fall below the performance of classic C++ operator overloading. The reason is that the resulting *Blitz++* and *uBLAS* kernels calculate one matrix element after another and traverse the right-hand side matrix column-wise (see also [2]). Listing 10 shows the C++ implementation of the multiplication operator, which does not contain any optimizations except for a suitable ordering of the nested `for` loops. The `Matrix` class represents a straightforward two-dimensional matrix data structure.

Listing 10: C++ implementation of the multiplication operator.

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

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

   return C;
}
```

Comparing the implementations in Listings 11 and 12, the advantage of the kernel encapsulation becomes apparent: whereas the BLAS function call is highly dependent on the matrix properties, changes to the matrix sizes, padding, or storage order do not affect the matrix multiplication in the *Blaze* library.

Listing 11: Row-major matrix/matrix multiplication via a call to the `cblas_dgemm` function.

```
double* A, *B, *C;
// ... Initialization of the matrices
cblas_dgemm( CblasRowMajor,
             CblasNoTrans,
             CblasNoTrans,
             N, N, N, 1.0, A, N, B, N, 0.0, C, N );
```

Listing 12: Row-major matrix/matrix multiplication in the *Blaze* library.

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

Figure 6 shows the result for a column-major matrix multiplication. Next to Intel's MKL, also *MTL4*, *Armadillo*,

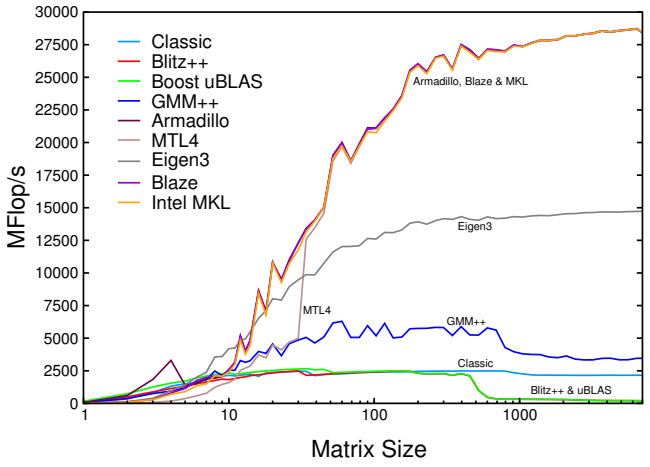

Figure 6: Performance of column-major matrix multiplication ($C^T = A^T * B^T$)

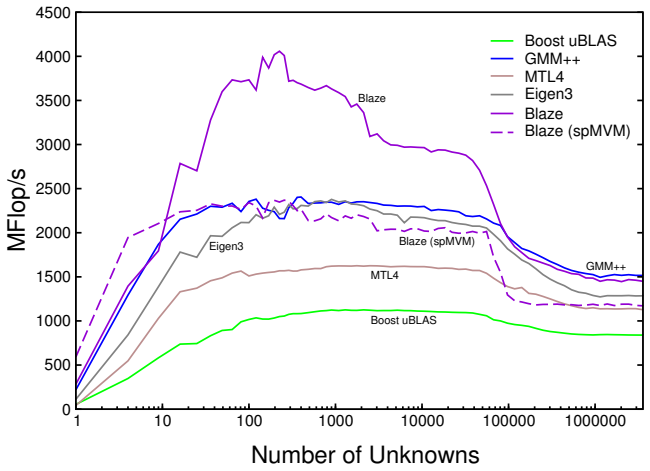

Figure 7: Performance of the CG algorithm vs. number of unknowns.

Eigen3, and *Blaze* achieve peak performance for large matrices. Whereas *Blitz++* and Boost *uBLAS* again fall below the performance of classic C++ operator overloading, *GMM++*, which does not yet support the MKL but only ATLAS, achieves a slightly better performance due to a more suitable default kernel.

D. Conjugate gradient algorithm

Listing 13: Implementation of the CG algorithm in the *Blaze* library.

```
using namespace blaze;

CompressedMatrix<double,rowMajor> A( N, N );
DynamicVector<double,false> x( N ), b( N ), r( N ),
                            d( N ), h( N );
double alpha, beta, delta;

// ... Initialization of the matrix and vectors

r = A * x - b;
delta = trans(r) * r;
d = -r;

for( size_t it=0UL; it<iterations; ++it )
{
   h = A * d;
   alpha = delta / ( trans(d) * h );
   x += alpha * d;
   r += alpha * h;
   beta = trans(r) * r;
   // omitted for benchmarking
   // if( sqrt( beta ) < threshold ) break;
   d = ( beta / delta ) * d - r;
   delta = beta;
}
```

A major advantage of (S)ET libraries is the intuitive, natural syntax, which enables the implementation of mathematical algorithms very close to the textbook presentation. Listing 13 shows the implementation of the CG algorithm using the *Blaze* library. Since CG appears in many applications, we use it to analyze the application performance of (S)ET libraries. As a toy problem we solve the Poisson equation ($\Delta u = 0$) on a two-dimensional (regular) grid with up to 2000^2 unknowns and Dirichlet boundary conditions. Although in this special case the structure of the grid could be exploited, we still use a sparse matrix to store the stencils in order to generalize the problem. The sparse matrix is stored in the common compressed row storage (CRS) format, which provides best performance for general matrices on standard processors [14]. Figure 7 summarizes the performance results. We only measure the time required for solving the given problem, restricting the number of iterations to either a maximum runtime of 10 seconds or the number of iterations required to solve the problem. The explicit convergence check is omitted for simplicity.

Listing 14: spMVM kernel ($C^b = A * x$).

```
for( size_t i=0UL; i<N; ++i ) {
   for( size_t j=row_ptr(i); j<row_ptr(i+1); ++j )
      b[i] = b[i] + val(j) * x[col_idx[j]];
}
```

Although the CG algorithm contains several BLAS-1 operations, a large part of the runtime goes into the sparse matrix-vector multiplication (spMVM). Listing 14 shows the spMVM kernel, which will not be vectorized due to the indirect access to the right-hand side vector (it would be easily possible here if we had exploited the matrix structure, but not in the general case).

The in-cache performance of the scalar spMVM kernel is limited by the accumulation into the LHS result: The compiler-generated code does not use modulo unrolling and thus has a loop-carried dependency, which leads to a three-cycle latency for every add operation. At 3.9 GHz, the in-L1 upper performance bound is thus 2/3 flops/cycle, i.e., 2.6 GFlop/s. The dashed line in Fig. 7 shows that *Blaze* achieves over 90% of this limit.

The other kernels in the CG algorithm are of daxpy type, scalar products, or vector norms. They are all SIMD-vectorizable and run at close to half-peak or peak performance with *Blaze* (see above; 80% of peak can be achieved for he vector norm). Hence, the spMVM is the performance-limiting

factor in the CG algorithm. The impact of the high-latency divides (about 20 cycles each) depends on the size of the problem; at 200 unknowns, where all arrays still fit into the L1 cache, one iteration of the whole CG solver takes about 808 ns, so the divides are negligible. Under these conditions we expect a performance of 4.6 GFlop/s, of which *Blaze* achieves about 87%. The other libraries are all significantly slower, mainly because of their inferior performance for the BLAS 1 kernels.

For large problem sizes an estimation of the achievable performance must be based on the main memory bandwidth. The difference between *GMM++* and *Blaze* can be explained by the different data type used for the sparse matrix indices: *Blaze* uses an eight-byte integral data type for the indices (`row_ptr` and `col_idx` in Listing 14), whereas *GMM++* uses only four bytes. Hence, the latter has almost 30% smaller data transfer requirement for the sparse matrix, which translates into about 12% for the whole CG solver (the code balance is 11.6 bytes/flop vs. 10.4 bytes/flop overall, leading to an expected performance of 1.59 GFlop/s and 1.77 GFlop/s, respectively). The reason why *GMM++* is only slightly faster than *Blaze* is its much smaller in-cache performance, which does not allow full exploitation of the memory bandwidth with a single thread.

It should be noted that it is possible to further improve the performance of the CG algorithm by fusing loops and therefore reducing memory traffic. However, this optimization cannot be performed by (S)ET-based libraries since it is not possible to optimize expressions across statements. Therefore although (S)ET libraries offer a very convenient way to implement a CG algorithm and provide perfect readability and with this also usability and maintainability, in order to achieve the maximum achievable performance it is still necessary to manually optimize the CG algorithm.

V. Conclusion and Future Work

By comparing several SET-based C++ math libraries we have demonstrated that it is possible to achieve a combination of HPC-grade performance with an elegant, intuitive usage that greatly benefits software development metrics such as maintainability, readability, programmability, and extensibility. Especially the *Eigen3* and *Blaze* libraries allow for easy, robust, and quick code development while still exploiting the performance of given hardware to a maximum.

At the time of writing, *Blaze* is the only library that allows for a full exploitation of the AVX instruction set on modern x86 processors, with a performance that comes very close to first-principles performance models. This result also demonstrates the significance of cooperation even within computer science between low-level performance engineers and high-level software architects to achieve a maximum in performance and usability.

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] T. Veldhuizen, "Expression Templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, 1995.

[2] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, "Expression Templates Revisited: A Performance Analysis of Current ET Methodologies," *Accepted for publication in SIAM Journal on Scientific Computing (SISC)*, 2012.

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

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

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

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

[7] Armadillo C++ linear algebra library, "Hompage of the Armadillo library: http://arma.sourceforge.net/."

[8] GMM++ library, "Homepage of the GMM++ library: http://download.gna.org/getfem/html/homepage/gmm.html."

[9] Automatically Tuned Linear Algebra Software (ATLAS), "Homepage of the ATLAS framework: http://math-atlas.sourceforge.net/."

[10] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[11] Blaze library, "Homepage of the Blaze library: http://www.zisc.uni-erlangen.de/blaze."

[12] J. D. McCalpin, "STREAM: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, VA, Tech. Rep., 1991-2007, a continually updated technical report. [Online]. Available: http://www.cs.virginia.edu/stream/

[13] J. Treibig and G. Hager, "Introducing a performance model for bandwidth-limited loop kernels," in *Proceedings of the Workshop "Memory issues on Multi- and Manycore Platforms" at PPAM 2009, the 8th International Conference on Parallel Processing and Applied Mathematics, Wroclaw, Poland*, 2009.

[14] G. Schubert, G. Hager, and H. Fehske, "Performance limitations for sparse matrix-vector multiplications on current multi-core environments," in *High Performance Computing in Science and Engineering, Garching/Munich 2009*. Springer Berlin Heidelberg, 2010, pp. 13–26.