Serdica J. Computing **4** (2010), 349–370

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

EXTENSION OF THE C-XSC LIBRARY WITH SCALAR PRODUCTS WITH SELECTABLE ACCURACY*

Michael Zimmer, Walter Krämer, Gerd Bohlender, Werner Hofschuster

Abstract. The C++ class library C-XSC for scientific computing has been extended with the possibility to compute scalar products with selectable accuracy in version 2.3.0. In previous versions, scalar products have always been computed exactly with the help of the so-called long accumulator. Additionally, optimized floating point computation of matrix and vector operations using BLAS-routines are added in C-XSC version 2.4.0. In this article the algorithms used and their implementations, as well as some potential pitfalls in the compilation, are described in more detail. Additionally, the theoretical background of the employed DotK algorithm and the necessary modifications of the concrete implementation in C-XSC are briefly explained. Run-time tests and numerical examples are presented as well.

1. Introduction and Notation. C-XSC [10, 8] is a C++-class library for verified scientific computing.In addition to basic types for calculating with real and complex (floating point) data, C-XSC also provides corresponding

ACM Computing Classification System (1998): G.1.0, G.4.

Key words: DotK algorithm, error-free transformations, C-XSC, scalar products, long accumulator, K-fold accuracy.

*Preliminary version of this paper was presented at the International Workshop on Mathematical Modelling and Scientific Computations (MMSC'09), September 23-26, 2009, Velingrad, Bulgaria.

matrix and vector data types and in particular data types and algorithms for interval arithmetic.

For all calculations of scalar products, which may be explicit or implicit (in the relevant operators), the so-called long accumulator [13, 14, 5] is used and is realized in the `dotprecision` data types. In the accumulator all scalar products of vectors with floating point components are calculated exactly in a sufficiently long fixed-point representation. With that, all calculations relevant for scalar products can be accomplished exactly. Only the final result has to be rounded into working precision. However, unless supported by hardware, such a scalar product calculation with the long accumulator is very time-consuming and is the main reason for the low run-time performance of C-XSC, e.g. in comparison to Intlab [18].

The goal of the changes described in this article is to introduce the selectable accuracy K (with respect to `double`-accuracy) for scalar products. Depending on the chosen accuracy, a different algorithm will be used (long accumulator, normal floating-point calculation, DotK algorithm [16]). To assure the compatibility with older programs, the changes were conducted in such a way that without any special declaration the case $K = 0$ will be assumed. This means that, the long accumulator is used, which is consistent with the behavior of older C-XSC versions.

With these changes, programs can now be better adapted to a particular task. This means that, calculations which don't require maximal accuracy can now be executed considerably faster, depending on the chosen accuracy K. Additionaly, pure floating-point calculations based on optimized BLAS routines are also available for some operations, leading to drastically improved run time performance in certain situations.

Let $\mathbb{F}$ denote the set of floating-point numbers and $\mathbb{F}^n$ the set of floating-point vectors of length n. The real operations $+, -, \cdot$ have to be replaced with the floating-point operations $\boxplus, \boxminus, \boxdot$. If a real-valued term E has to be calculated in floating-point arithmetic, we denote it in the short form $\mathrm{fl}(E)$. E. g. $\mathrm{fl}(\sum_{i=1}^{n-1} p_i + p_n)$ denotes the floating-point evaluation of the term $\sum_{i=1}^{n-1} p_i + p_n$, which is the result of the calculation of $p_1 \boxplus p_2 \boxplus \ldots \boxplus p_n$ (all operations of the real-valued term are replaced by the corresponding floating-point operations). This article assumes the double format of the IEEE standard [2] as the underlying floating point format. *eps* denotes the relative error bound of rounded floating-point operations; for IEEE double operations, $eps = 2^{-53}$ holds.

The article is structured as follows. First, the current status with regard to explicit as well as implicit (i. e. hidden in operator calls) scalar product calculations in C-XSC is clarified in simple examples. In section 3 the theoretical background of the DotK algorithm is briefly explained. Section 4 describes the actual implementation, especially the modifications which were applied to the DotK algorithm and how and where optimized BLAS routines are used. In section 5 some important points concerning compilation are highlighted, especially regarding performance and numerical accuracy. Finally, in section 6 some examples for the usage and also time measurements are presented.

2. Computation of Exact or Maximally Accurate Scalar Products in C-XSC. The following code in C-XSC

```
//Real matrices
rmatrix A(m,n); //m-by-n
rmatrix B(n,p); //n-by-p
rmatrix R(m,p); //m-by-p

//..matrices A and B are
//initialized with double values...

//maximally accurate matrix-matrix
//multiplication
R= A*B;
```

computes the result `R` of the multiplication of the real floating point matrix `A` with the real floating point matrix `B`. Each element of the resulting matrix `R` is computed with maximal accuracy. For the calculation of the end result with maximal accuracy, internally, the scalar product of the corresponding row of matrix `A` with the corresponding column of matrix `B` is calculated componentwise exactly(!) in the so-called long accumulator. The exact result is rounded to the nearest floating point number (maximally accurate floating point result). This long accumulator is realized with the data type `dotprecision` offered in C-XSC, which is also directly available to the user. The scalar product of two floating-point vectors can thus be calculated exactly as follows:

```
rvector x(n);
rvector y(n);

//...components of x and y are
//initialized with double values...

//accu is initialized with 0
dotprecision accu(0);
```

```
accumulate(accu,x,y);
//exact value of the scalar product
//x*y is available in accu
```

The leading `r` in the name of the data types `rvector` and `rmatrix` indicate that their element are real.

After execution of this program fragment, the `dotprecision` variable `accu` contains the exact value of the scalar product of the two floating point vectors `x` and `y`. The C-XSC routine `accumulate` allows to exactly add the exact value of the scalar product of vectors `x` and `y` to the previous value which is already stored in the `dotprecision` variable `accu` (here, this is 0). In this computation, neither overflow nor underflow can occur (but this can, of course, happen in a later assignment of the (rounded) exact value to a floating-point variable). The exact value of such a scalar product is typically not a floating-point number.

The described feature of the exact calculation of scalar products of floating-point vectors is used in C-XSC to implement all basic operations with maximum accuracy. This applies also to all matrix and vector operations, to complex operations and to operations with floating-point intervals. The long accumulator (`dotprecision` data type) is right now unfortunately not supported by any special hardware. Thus, the long accumulator is implemented in software as a fixed-point format with sufficient mantissa length, so that operations which use this data type are relatively slow (e.g. matrix-matrix multiplication).

In order to improve the run-time of C-XSC programs two modifications are introduced. First, so-called *error-free transformations* are used to accelerate the higher-precision scalar products (compared to the software solution of the long accumulator). This is the central theme in this article. The second step is to use highly optimized BLAS-routines for some floating-point operations (especially matrix-matrix multiplications), which will also be elaborated on later in this paper.

The DotK algorithm described in the following section uses only floating point operations for calculations of scalar products of floating point vectors in a simulated higher precision. This algorithm is significantly faster compared to scalar product calculations with a software emulation of the long accumulator. In contrast, the hardware realization of the long accumulator (which is possible with moderate additional hardware cost [15]) would be significantly faster than the DotK algorithm. It should also be noted that the DotK algorithm normally does not guarantee matrix-vector-operations with maximum accuracy. A hardware solution of the long accumulator would always be maximally accurate. The

availability of the DotK algorithms in C-XSC is motivated only by the unfortunate **non-**availability of the long accumulator in hardware.

3. The DotK-Algorithm. The DotK algorithm for the fast calculation of scalar products of floating point vectors in simulated higher precision (K-fold double accuracy is simulated) was introduced in original form in [16, 20]. It is based on the use of error-free transformations: For all $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \cdot\}$ there exists a $y \in \mathbb{F}$ with $a \circ b = x + y$ and $x = fl(a \circ b)$.

Each of the operations occurring in a scalar product can be converted exactly into the sum of two floating point numbers x and y, where x is the result of the normal floating point calculation and y represents the emerging rounding error. This also applies whenever (intermediate) results are produced in the underflow range. Each scalar product of vectors with floating point components can thus be transformed error-free into a sum of floating point numbers.

The calculation of error-free transformations is possible with mere floating-point operations [7, 6, 16, 20, 19]. Error-free transformations were already used in the 1970s in order to compute scalar products with higher accuracy [17] or even with maximum accuracy [4].

The algorithm 1 (`TwoSum`) is used for the transformation of the sum of two floaing point numbers, the algorithm 3 (`TwoProduct`) is used for the tranformation of the multiplication. It utilizes algorithm 2 (`Split`), which splits a floating point number into the sum of two floating point numbers with non-overlapping mantissas.

Input : Two floating-point numbers $a, b \in \mathbb{F}$
Output: Two floating-point numbers $x, y \in \mathbb{F}$ with $x = a \boxplus b$ and $a + b = x + y$
$x = a \boxplus b$
$z = x \boxminus a$
$y = (a \boxminus (x \boxminus z)) \boxplus (b \boxminus z)$

Algorithm 1: TwoSum

These algorithms can be implemented directly in C++, where `factor` of the algorithm `Split` should be implemented as a constant. In the implementation extreme care should be taken that all calculations are conducted according to the IEEE standard for `double` precision. In particular, many modern processors use 80 bit wide registers for floating point calculations. The usage of such excess precision registers can lead to wrong results during the error-free transformations. Details on this topic can be found in section 5.

Input : A floating-point number $a \in \mathbb{F}$
Output: Two floating-point numbers $x, y \in \mathbb{F}$ with $a = x + y$
$factor = 2^{27} \boxplus 1$
$c = factor \boxdot a$
$x = c \boxminus (c \boxminus a)$
$y = a \boxminus x$

Algorithm 2: Split

Input : Two floating-point numbers $a, b \in \mathbb{F}$
Output: Two floating-point numbers $x, y \in \mathbb{F}$ with $a \cdot b = x + y$
$x = a \boxdot b$
$[a_1, a_2]$ =Split(a)
$[b_1, b_2]$ =Split(b)
$y = a_2 \boxdot b_2 \boxminus (((x \boxminus a_1 \boxdot b_1) \boxminus a_2 \boxdot b_1) \boxminus a_1 \boxdot b_2)$

Algorithm 3: TwoProduct

Based on the error-free calculations, the summation of the single elements of a vector of floating-point numbers can be implemented in K-times working precision (algorithm 4, `SumK`). The principle in doing so is to implement the summation with the help of `TwoSum`, carrying the floating point result along and saving the error-terms as the corresponding element of the vector. With each run, the condition number improves by almost a factor *eps* (relative rounding error). After $K - 1$ repetitions and a final summation in normal floating point, K-times accuracy is achieved.

Now the actual algorithms for the computation of the scalar product in K-fold accuracy can be formulated. For this purpose the scalar product is converted

Input : A vector $p \in \mathbb{F}^n$, desired accuracy K
Output: The sum $fl(\sum p_i)$, in K-fold working accuracy
for *k = 1 : K-1* **do**
  for *i = 2: n* **do**
    $[p_i, p_{i-1}]$ = TwoSum(p_i,p_{i-1})
res = fl$(\sum_{i=1}^{n-1} p_i + p_n)$

Algorithm 4: SumK

Input : Two vectors $x, y \in \mathbb{F}^n$
Output: The scalar product $res = x \cdot y$, as if calculated with double working accuracy
[p,s] = TwoProduct(x_1,y_1)
for *i=2:n* **do**
  [h,r] = TwoProduct(x_i,y_i)
  [p,q] = TwoSum(p,h)
  s = s ⊞ (q⊞r)
res = p ⊞ s

Algorithm 5: Dot2

Input : Two vectors $x, y \in \mathbb{F}^n$, desired accuracy K
Output: The scalar product $x \cdot y$, as if calculated with K-times working accuracy
[p,r_1] = TwoProduct(x_1,y_1)
for *i=2:n* **do**
  [h,r_i] = TwoProduct(x_i,y_i)
  [p,r_{n+i-1}] = TwoSum(p,h)
r_{2n} = p
res = SumK(r, K-1)

Algorithm 6: DotK

into a sum of floating point numbers with the help of the error free transformations, which can then be calculated in K-fold precision with the algorithm `SumK`. In the case of $K = 2$, an optimized version of the algorithm (algorithm 5, `Dot2`) is used, because it is not necessary to store the error-terms of the error-free transformations in an array or a vector. For $K > 2$ the algorithm 6 (`DotK`) will be used.

It is also possible to determine an error bound for these algorithms via mere floating-point calculation. This method is not further illustrated here, more detailed information can be found in [16]. The necessary calculations are specified in section 4 in the exemplification of the concrete implementation. In this section, some modifications of the original version of the DotK algorithm (described above) are enlarged on, which essentially serve for a reasonable integration in the accumulator classes.

In [12, 21] a more detailed summary of the DotK algorithm can be found, as well as the description of an earlier implementation for C-XSC in the form of

an additional package of separate classes. It is also possible to convert the result of a dot product to `staggered precision`, this is described in [3].

4. Implementation. In this section, the changes to the C-XSC library and the modifications to the algorithm from section 3 are described in more detail. The changes were implemented in such a way that existing programs behave exactly the same as with previous C-XSC versions. Absolutely no changes in the code should be necessary to retain compatibility.

4.1. Changes to the `dotprecision`-Classes. Since the accuracy for the calculation of scalar products should now be selectable (i. e. switchable at run time), the `dotprecision` classes need a new member variable for the currently chosen accuracy. For this the following changes are conducted in all `dotprecision`-classes:

- A new member variable `int k` to store the current accuracy with the following meaning:
  - $k = 0$: Calculation with accumulator (as in old C-XSC versions).
  - $k = 1$: Pure floating-point calculations (enclosure or error-bound is calculated via switching of the rounding mode).
  - $k \geq 2$: Use of the DotK algorithm for calculation in K-fold accuracy.
- New member functions `void set_k(unsigned int)` and `int get_k()` for setting and reading-out the current accuracy level.
- Corresponding adjustment of the constructors (accuracy is by default set to 0, in the copy-constructor the accuracy of the original is chosen) and assignment operators (accuracy is **not** assigned but always retained).

In addition, an error variable `err` of type `real` for the class `dotprecision` is required, because now the currently saved value does not need to be exact anymore. During the call of the function `rnd` to round the result to `double` precision this error bound is now taken into account. The same applies to comparison operators. All other `dotprecision` classes do not need such an error variable, because they are composed of real `dotprecision` objects (the class `idotprecision` uses e.g. two `dotprecision` objects for representing the infimum and the supremum of the resulting interval, respectively).

4.2. Implementation of the new Computation Functions. For the actual computation of the scalar product, a new function (or rather several overloaded versions) `addDot(dotprecision&, const S&, const T&)` is introduced,

which is available only internally. This function is implemented as a template with parameters `S` and `T`. This is done to directly support auxiliary data types of C-XSC such as `rvector_slice` which are used for cutting out slices of vectors and matrices. Thus, one does not have to write an adapted version of the function for each auxiliary data type (since corresponding data types have the same set of operators, the source code for the computation would be identical).

This function, with all required auxiliary functions (implementation of the error-free transformations as well as the algorithm `SumK`, which is required by the DotK algorithm for $K > 2$), is implemented in a new file `dotk.inl` (and `idotk.inl`, `cdotk.inl` and `cidotk.inl` for the appropriate data type). The declarations of the function do not appear in any header-file. They can thus not be called by the user of the library directly (instead, the user calls the `accumulate` function, which in turn calls `addDot` if necessary).

These new files have to be included in the `.cpp`-files in which the scalar product calculations are required (see section 4.3). In this way the code for the computations is compiled directly into the library, the definitions are not, as usual in C-XSC, embedded into the header. This is required because in many cases special compiler flags are needed to guarantee the correct execution of error-free transformations. Otherwise, the user would have to set these flags in each compilation of a C-XSC program using the DotK algorithm to guarantee correct results (see also section 5).

Listing 1: Computation function for real scalar products

```
template<typename S, typename T>
inline void addDot(dotprecision &val,
                   const S &x, const T &y) {
int n = Ub(x)-Lb(x)+1;
int lb1 = Lb(x);
int lb2 = Lb(y);
real res,err=0.0;
int rnd;

//Check rounding mode
if((rnd=getround()) != 0) {
        setround(0);
}

if(val.k == 0) { //use accumulator
        for(int i=1 ; i<=n ; i++)
                accumulate(val,x[i+lb1-1],y[i+lb2-1]);
} else if(val.k == 1) { //use floating point
        real resd = 0.0, resu = 0.0;

        setround(-1);
        for(int i=1 ; i<=n ; i++)
                resd += x[i+lb1-1] * y[i+lb2-1];
```

```
        setround(1);
        for(int i=1 ; i<=n ; i++)
                resu += x[i+lb1-1] * y[i+lb2-1];

        setround(0);
        res = resd+(resu-resd)*0.5;
        setround(1);
        val.err = val.err + resu-res;
        val += res;
} else if(val.k == 2) { //use DotK optimized for K=2
        real p, s, h, r, q, t;

        TwoProduct(x[lb1],y[lb2],p,s);

        err += abs(s);

        for(int i=2 ; i<=n ; i++) {
                TwoProduct(x[lb1+i-1],y[lb2+i-1],h,r);
                TwoSum(p,h,p,q);
                t = q + r;
                s += t;
                err += abs(t);
        }

        val += p;
        val += s;
        res = p+s;

        real alpha, delta, error;

        delta = (n*Epsilon) / (1.0-2*n*Epsilon);
        alpha = (Epsilon*abs(res)) +
                (delta*err+3*MinReal/Epsilon);
        error = alpha / (1.0 - 2*Epsilon);

        setround(1);
        val.err = val.err + error;
} else { //use DotK
        real r = 0, h;
        real* t = new real[2*n];

        for(int i=1 ; i<=n ; i++) {
                TwoProduct(x[lb1+i-1], y[lb2+i-1],
                                h, t[i-1]);
                TwoSum(r, h, r, t[n+i-2]);
        }

        t[2*n-1] = r;
        SumK(t, 2*n, val.k-1, err, val);

        setround(1);
        val.err = val.err + err;

        delete[] t;
}

//Reset rounding mode to former value
setround(rnd);
}
```

Listing 1 shows the implementation of `addDot` for the real scalar product. According to the value of the member variable `k` of the `dotprecision` object, an appropriate calculation method is used. The case $K = 2$ shows how the error bound can be computed with simple floating-point calculations. For $K > 2$ the error bound is calculated in a similar manner in the `SumK` function.

If no interval scalar product is calculated, a special variant of `addDot` exists in which the computation of the error bound is omitted. This version is used in the real or complex operators of the matrix and vector classes (see section 4.4), since these operators do not compute enclosures and thus need no error bound. That way, the execution time can be reduced. This variant without error calculation is also directly available for the user in the form of the function `accumulate_approx` (as described, this function is **not** available for interval scalar products, since then correct enclosures are expected in general).

An important modification compared to the original algorithm is made possible through the usage of the accumulator. Since the result of every scalar product calculation must be saved intermediately in the respective accumulator anyway, one can take advantage of its ability to store very long numbers exactly. The result may be calculated in such a way that the data in the accumulator can be converted to a staggered representation later on, if necessary. For this purpose, the current error term (the sum of the error terms of the single operations) is individually added to the accumulator in every run of the DotK algorithm. This error term decreases by a factor of *eps* in every run. Finally, at the end of the calculation, the computed end result is added. The remaining error terms are used to compute the final error. Thus, the result stored in the accumulator has K-fold double length. Together with the error bound, all necessary information for a staggered representation of the result is stored in the `dotprecision` object. For intervals and complex numbers, equivalent implementations are used. This topic is described in more detail in [3].

For $K = 1$, i. e. pure floating-point calculation, the normal operators of the class `interval` are not used in the real interval case, since this would be very slow due to the frequent switching of the rounding mode. Instead, the vectors for the scalar products to compute the infimum and the supremum the result vector generated according to the interval multiplication chart. The scalar product for the infimum and supremum of the result can then be calculated separately.In this way, in total, the rounding mode has to be switched only twice. Complex intervals use midpoint-radius representation for performance reasons [18]. However, the real part and the imaginary part are computed separately, so no complex disc arithmetic is used.

4.3. Adjustment of the Calls of `accumulate`. The actual computation of the scalar product is always started with a call of an `accumulate` function, both for direct calculation with the `dotprecision` data types and also for the usage of an appropriate operator. Due to the fact that the calculation source code has to be directly compiled into the library for the abovementioned reasons, the calls of `accumulate` can no more be carried out "inline".

But all newly added functions, especially `addDot`, are declared as `inline`, which means the function call in the `accumulate` function is for free in the best case. Unfortunately, the call of `accumulate` itself can no more be carried out "inline" for the mentioned reasons. Time measurements show that this seems to have only minor consequences on the execution time in general.

4.4. Adjustment of the Operators with Implicit Scalar Products and BLAS support. In many operators of the vector and matrix classes, one or many scalar products are implicitly calculated. These calculations have to be adapted to the previous changes. For this reason a new global variable `opdotprec` is introduced (analogous to `stagprec` when using staggered arithmetic). This variable indicates the desired accuracy of the scalar products, which are calculated during the call of appropriate operators.

During the call of the operator, a local variable of type `dotprecision` is now created, whose accuracy is set to the current value of `opdotprec`. With the help of this variable all occurring scalar products can be calculated.

If precision `opdotprec=1` is selected, the operators can also be computed using BLAS-routines. For this, the compile switch `CXSC_USE_BLAS` has to be set during compilation and the program hast to be linked to an appropriate BLAS library and the CBLAS interface library. The computations of the operator are then computed not by calls to `accumulate`, but by converting the C-XSC data types into appropriate BLAS arrays and calling the corresponding BLAS routine.

For intervals, manipulation of the rounding mode of the processor is used to compute reliable enclosures. Also, in some cases, especially for products of interval matrices, midpoint radius representation is used to save run time. The algorithms used in this case are similar to the ones described in [18].

With these changes all scalar products in C-XSC can now be calculated in selectable accuracy, either by setting the member variable `k` of the `dotprecision` data types for the direct calculation or by setting the global variable `opdotprec` for the calculation with operators. An example is shown in section 6. When `double` precision is sufficient, optimized BLAS routines can now be used, which can result in huge speed-ups. Also, most BLAS routines are already multithreaded, so the performance gain on multicore or multiprocessor machines may be even higher.

5. Remarks on the Compilation For the compilation of the library some important aspects have to be taken into consideration, because on one hand the performance of the DotK calculations can heavily collapse, and on the other hand even totally wrong results can emerge, if the compiler flags are not set appropriately. In the following some remarks will be given on important points regarding performance and accuracy during the compilation.

5.1. Performance. A really crucial factor for the performance is – as for the whole C-XSC library – the activation of inlining and its corresponding application through the compiler. In the past, inlining in C-XSC had to be activated by setting the compiler switch `_CXSC_INLINE`. Starting with version 2.3.0, inlining is activated by default.

To be able to use the speed advantages due to inlining, appropriate compiler options have to be set. For this, either compilation with full optimization (`-O3`) is required or inlining has to be activated separately (`-finline-functions`). (This refers to the GNU-compiler, other compilers might activate inlining at other optimization levels or not at all without explicit setting of the switch.) However, in general at least the optimization level `-O1` in connection with inlining should be set for the compilation of the DotK algorithms, in order to achieve an acceptable performance. The optimization level is selectable during the installation of C-XSC with the corresponding installation script.

The compiler version is also of great importance. In GCC version 4.3.x, inlining support is innately much better than in version 4.0.x. This is due to the considerably higher standard limits for the allowed size of inline functions. In older versions the limits can be increased by setting some parameter (see [1], section Command Options / Options That Control Optimization), which can possibly lead to better results. However, the correct setting of the limits is not trivial (the compilation time as well as the code size can increase dramatically if these limits are set liberally). Generally, it is recommended to use a newer version of the compiler. Similar considerations should also apply to other compilers.

5.2. Numeric Accuracy. Many modern processors, especially nearly all Intel processors, use 80 bit wide registers for floating-point calculations, i. e. they internally use a higher accuracy than the IEEE standard defines for the data type `double`. With activated compiler optimization, intermediate results during the computation of the error-free transformations are kept in the registers, because this offers enormous speed advantages compared to writing the intermediate results back into the main memory or cache.

However, under these conditions the algorithms for the error-free transformations may deliver incorrect results, because they require accurate compliance

with the IEEE standard. If within the compilation of the DotK-code no compiler switch is used which enforces this behavior, the program might compute completely wrong results.

The best option on modern x86 processors is the usage of SSE registers (Streaming SIMD Extensions) for floating-point calculations. The SSE instruction set extension was introduced by Intel for SIMD calculation (single instruction, multiple data) around the turn of the millennium, especially with regard to multimedia application. Since the introduction of the SSE2 instruction set, this extension also supports computations with `double` values in special registers which are 128 bits wide (one register is able to hold two `double` values and can apply a calculation at the same time). This instruction set with corresponding registers is included in every x86 processor since to Pentium 3 (the same applies for AMD).

For the usage of the SSE-registers for floating-point calculations a compiler switch normally has to be activated:

- For GCC: The Option `-mfpmath=sse` for the usage of the SSE instruction set for floating-point calculations as well as the option `-msse2` for the activation of the SSE2 instruction set should be activated (in newer processors, the newer versions of the instruction set should be activated, e.g. `-msse3`). When 64 bit code is generated, these options are activated by default.
- For the Intel-compiler: Option `-msse2` (or `-msse3` etc.) has to be activated, floating-point calculation will be automatically adapted. The activation of this option also activates auto-vectorization, which means that the compiler tries to optimize parts of the code, like for-loops, for the SIMD instructions which may increase the performance. Similar optimizations are also possible in GCC since version 4.4.0.

If the processor uses excess precision, but does not support the SSE instruction set (or an equivalent solution), a different option has to be activated. The Intel compiler offers the possibility to control the accuracy and the speed of floating-point calculation with the option `-fp-model` [9]. During the usage of `-fp-model source` e.g. the accuracy given in the source code (here `double`) is strictly maintained. This normally leads to performance losses, since the intermediate results are not maintained in the registers but are stored in main memory.

An alternative version in GCC is `-ffloat-store`, with which all intermediate results are written back to the memory. This also causes dramatic performance losses.

The relevant compiler switches are set automatically by the installation script on many platforms. If not, they have to be added by hand during the optimization option, or the Makefile has to be adjusted accordingly.

6. Examples and Time Measurements. In the following, a short example for the usage of the changed C-XSC library will be given. Then some time measurements will follow demonstrating which performance and which accuracy can be expected with the correct compilation of the library according to section 5 and the remarks presented here.

6.1. Example. Listing 2 shows a small example which clarifies the usage of the new scalar products with corresponding comments.

Listing 2: Code Example

```
int n=1000;
rvector x(n), x2(n);
ivector y(n);
rmatrix A(n,n);
imatrix B(n,n);
interval z;

//fill vectors and matrices ....

dotprecision dot(0.0);
idotprecision idot(0.0);

//set accuracy for scalar products
//in operators
opdotprec = 1;

//dot calculates with two-fold double accuracy
dot.set_k(2);
//idot calculates with maximal accuracy (accu)
idot.set_k(0);

//calculate z in floating point, since opdotprec=1
//(uses BLAS if CXSC_USE_BLAS is set)
z = x*y;

opdotprec = 3;

//is calculated in three-fold accuracy
B = A*y;

//calculates x2=A*x in two-times accuracy
//without error-bound
for(int i=1 ; i<=1000 ; i++) {
    dot = 0.0;
    accumulate_approx(dot, A[i], x);
    x2[i] = rnd(dot);
}
```

```
//calculates y = B*x in maximal accuracy
for(int i=1 ; i<=1000 ; i++) {
    idot = 0.0;
    accumulate(idot, B[i], x);
    y[i] = rnd(idot);
}
```

6.2. Time Measurements. The following output was produced by a test program which can be downloaded from

`http://www.math.uni-wuppertal.de/~xsc/cxsc/examples/#SJC1`

For real scalar products on a machine with two Intel Xeon 2.26 GHz processors (Nehalem architecture) and 24GB RAM the listed results were achieved (however, since no threading was used, only one core of one processor is involved in the computations). The GNU compiler version 4.4.1 was used with full optimizations. The accuracy and the relative speed should be similar on most other systems, when compiled correctly. The dimension of the scalar product is $n = 1\,000\,000$, the condition is 10^{100}, i. e. extremely ill conditioned, and every calculation was repeated 10 times.

```
Exact result: +1.0000000000000000E-100

k=0:
[+1.0000000000000000E-100,+1.0000000000000001E-100]
Is an enclosure of the correct result!
Time used: 0.890723s

k=1:
[-1.0667230210259506E-008,+1.0652418946932587E-008]
Is an enclosure of the correct result!
Time used: 0.0423551s

k=2:
[-9.9645319516326720E-020,+9.9645373537521509E-020]
Is an enclosure of the correct result!
Time used: 0.102189s

k=3:
[-1.0537382008712601E-033,+1.0537377990909645E-033]
Is an enclosure of the correct result!
Time used: 0.185942s

k=4:
[-1.6669881848134855E-047,+1.6669879319579374E-047]
Is an enclosure of the correct result!
Time used: 0.229s

k=5:
[-9.5915709584447162E-062,+9.5915715380081772E-062]
Is an enclosure of the correct result!
Time used: 0.286885s

k=6:
[-1.1797502641781892E-075,+1.1797502950635348E-075]
```

```
Is an enclosure of the correct result!
Time used: 0.356305s

k=7:
[-7.0011492809898146E-090,+7.0011493687712463E-090]
Is an enclosure of the correct result!
Time used: 0.404176s

k=8:
[+9.9999718505357947E-101,+1.0000028149462135E-100]
Is an enclosure of the correct result!
Time used: 0.458884s

k=9:
[+9.9999999999999989E-101,+1.0000000000000002E-100]
Is an enclosure of the correct result!
Time used: 0.515731s

k=10:
[+1.0000000000000000E-100,+1.0000000000000001E-100]
Is an enclosure of the correct result!
Time used: 0.573675s
```

Table 1 shows an overview over the measured run-times on the above system for corresponding scalar products with all four basic data types. Here the dimension is also $n = 1\,000\,000$ and all measurements were repeated ten times.

Table 1. Time measurement for scalar products with dimension $n = 1\,000\,000$ in seconds, each repeated ten times

K	**real**	**interval**	**complex**	**cinterval**
0	0.89	2.10	3.50	8.52
1	0.04	0.31	0.10	0.52
2	0.10	0.46	0.30	1.84
3	0.19	0.70	0.97	2.88
4	0.23	0.82	1.19	3.36
5	0.29	0.94	1.42	3.82
6	0.36	1.06	1.65	4.28
7	0.40	1.17	1.88	4.74
8	0.46	1.29	2.11	5.21
9	0.52	1.40	2.34	5.67
10	0.57	1.52	2.57	6.13

The results show that even with an accuracy of $K = 10$ the DotK algorithm is still faster on the test system than the old calculation method using the long accumulator. If the accuracy does not matter, e.g. in the calculation of approximate solutions, a speed increase by the factor of 10 to even 40 can be achieved by setting the accuracy to $K = 1$.

To demonstrate the effect of the BLAS routines, the time needed for a matrix-matrix product with two 500×500 matrices was measured for all four basic datatypes for precision 0 to 5. The results can be seen in Table 2. The Intel Math Kernel Library version 11.1 was used as BLAS library on the above mentioned systems. The number of threads for the BLAS routine was set to one so that again only one core was used in all computations.

Table 2. Time measurement for 500×500 matrix-matrix products

K	**real**	**interval**	**complex**	**cinterval**
0	10.66	23.54	41.93	107.6
1	0.65	3.16	0.81	6.78
1 (BLAS)	0.014	0.072	0.066	0.35
2	1.84	4.99	3.99	20.41
3	2.77	6.65	10.31	26.42
4	3.43	8.01	13.19	31.86
5	4.14	9.39	16.16	37.34

The results show that the optimized BLAS routines can significantly enhance run-time performance if no increased accuracy is needed (the times reported using BLAS for the cases of interval matrices and complex interval matrices comprise the time needed for intermediate conversions to midpoint/radius representation). The results for the other precisions are consistent with the measurements of the single scalar products.

7. Conclusion and Future Prospect. Scalar product calculations are an essential element of most numeric calculations. They should be as accurate and efficient as possible, or if both is not possible, the user should be able to decide how the priorities have to be set. The safe error bounds which are required for verified numeric methods are also delivered by the available routines [21, 12, 11, 3].

With the new possibilities in the calculation of scalar products, the C-XSC library gains considerable flexibility. Scalar products, which need not be executed with maximal accuracy, can be computed with a selected accuracy. Our run-time

measurements show clearly that remarkable speed increases can be achieved. Due to the fact that the accuracy requirements in scalar product calculations can be switched any time in the simplest way, the user has now the possibility to optimize his program towards numeric accuracy as well as towards high performance.

Possible extensions for the future are e.g. the usage of the new summation algorithm presented by Rump [19] which is faster than `SumK`, at least in theory.

Remark: The usage of the so-called DotK algorithm in C-XSC clearly leads to improved execution times. This is, however, only due to the regrettable fact that the long accumulator is still not supported in hardware on today's processors. With such commonly claimed hardware support, scalar products of floating point vectors could always be computed exactly and with optimal speed (that means clearly faster than with all known scalar product algorithms based on error free transformations, even if supported by hardware as well).

REFERENCES

[1] GCC. Online documentation.
`http://gcc.gnu.org/onlinedocs`

[2] ANSI/IEEE. Std. 754-1985, A Standard for Binary Floating-Point Arithmetic. New York, 1985, reprinted in SIGPLAN **22**(1987), No 2, 9–25.

[3] Blomquist F., W. Hofschuster, W. Krämer. A Modified Staggered Correction Arithmetic with Enhanced Accuracy and Very Wide Exponent Range. In: Lecture Notes in Computer Science, Vol. **5492**, Springer, 2009, 41–67.

[4] Bohlender G. Genaue Berechnung mehrfacher Summen, Produkte und Wurzeln von Gleitkommazahlen und allgemeine Arithmetik in höheren Programmiersprachen. Dissertation, Universität Karlsruhe, 1978.

[5] Bohlender G. What do we need beyond IEEE Arithmetic? Computer Arithmetic and Self-validating Numerical Methods, Academic Press, San Diego, 1990, 1–32.

[6] Bohlender G., W. Walter , P. Kornerup, D. W. Matula. Semantics for Exact Floating Point Operations. In: Proceedings of the 10th IEEE Symposium on Computer Arithmetic, Grenoble, 26-28 June 1991, 22–26.

[7] Dekker T. J. A floating-point technique for extending the available precision. *Numer. Math.*, **18** (1971), No 3, 224–242.

[8] Hofschuster W., W. Krämer. C-XSC 2.0: A C++ Library for Extended Scientific Computing.In: Numerical Software with Result Verification, Lecture Notes in Computer Science, Vol. **2991**, Springer-Verlag, Heidelberg, 2004, 15–35.

[9] INTEL. C++ Compiler User and Reference Guides.
`http://software.intel.com/en-us/intel-compilers`

[10] Klatte R., U. Kulisch, A. Wiethoff, C. Lawo, M. Rauch. C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Heidelberg, 1993.

[11] Kolberg M. Parallel Self-Verified Solver for Dense Linear Systems. PhD Thesis, PUCI, Porto Alegre, 2009.

[12] Krämer W., M. Zimmer. Fast (Parallel) Dense Linear System Solvers in C-XSC Using Error Free Transformations and BLAS. Lecture Notes in Computer Science, Vol. **5492**, Springer, 2009, 230–249.

[13] Kulisch U., W. Miranker. The arithmetic of the digital computer: A new approach. *SIAM Rev.*, **28** (1986), No 1, 1–40.

[14] Kulisch U. Die fünfte Gleitkommaoperation für Top-Performance Computer. Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und numerische Algorithmen mit Ergebnisverifikation, 1997.

[15] Kulisch U. Computer Arithmetic and Validity - Theory, Implementation and Applications. De Gruyter, Berlin, 2008.

[16] Ogita T., S. M. Rump, S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, **26** (2005), No 6, 1955–1988.

[17] Pichat M. Correction d'une somme en arithmétique a virgule flottante. *Numer. Math.*, **19** (1972), 400–406.

[18] Rump S. M. Intlab - Interval Laboratory. In: Developments in Reliable Computing (Ed. T. Csendes), Kluwer Academic Publishers, 1999, 77–104.

[19] Rump S. M. Ultimately Fast Accurate Summation. *SIAM Journal on Scientific Computing*, **31** (2009), No 5, 3466–3502.

[20] Yamanaka N., T. Ogita , S. M. Rump, S. Oishi. A Parallel Algorithm for Accurate Dot Product. *Parallel Computing*, **34** (2008), 392–410.

[21] Zimmer M. Laufzeiteffiziente, parallele Löser für lineare Intervallgleichungssysteme in C-XSC. Master Thesis, University of Wuppertal, 2007.

8. Appendix. A test program which measures the run-time of point and interval scalar product calculations (real and complex) can be downloaded from `http://www.math.uni-wuppertal.de/~xsc/cxsc/examples/#SJC2`.

The results give a first impression of the run-time savings to be expected depending on the used algorithm. They are as far as possible self-explaing. A penalty factor is printed for each specific kind of scalar product calculation. It refers to a simple C++ loop for the calculation of a scalar product of floating-point vectors, whose components are double precision numbers (this should be the fastest kind of calculation, but possibly with imprecise or even totally wrong numeric results).

Executing this program on an Intel Xeon 3.4GHz processor with 2GB RAM, the following output (shortened by hand) is produced:

```
Time measurements for different kinds of
dot product computations
Vector length: 100000
repMax: 100
baseTime using IEEE double: 0.076174
ds: 1.21578e+06
1)  Double ds+= dx*dy:
Penalty    1.0,  time used  0.0764107704
ds: 1215777.3156745557
2)  Double array ds+= dax[i]*day[i]:
Penalty    1.0,  time used  0.0774068832
ds: 1215789.4735692909
2b)  Double arrays created with new:
Penalty    1.0,  time used  0.0777688026
ds: 1215789.4735692909
3)  Real rs+= rx*ry:
Penalty    0.2,  time used  0.0147631168
rs: 1.215777E+006
4)  Real using rvector rs+= rvx[i]*rvy[i]:
Penalty    0.7,  time used  0.0536749363
rs: 1.215777E+006
5)  Interval is+= ix*iy:
Penalty   75.5,  time used  5.7542631626
is: [1.215777E+006,1.215778E+006]
6)  Dotprecision rdots+= rx*ry:
Penalty    7.4,  time used  0.5653319359
rdots: 1.2157773158E+0006
7)  Idotprecision with idots+= ix*iy:
Penalty   47.1,  time used  3.5909459591
9)  Dot product using rvectors and Dot1:
Penalty    0.7,  time used  0.0548717976
10) Dot product using rvectors and Dot2:
Penalty    2.3,  time used  0.1721401215
11) Dot product using rvectors
    and Dot0 (accu):
Penalty   15.8,  time used  1.2017669678
12) Interval dot product using
    ivectors and Dot1:
Penalty    6.9,  time used  0.5260379314
13) Interval dot product using
    ivectors and Dot2:
```

```
Penalty    8.5,  time used  0.6460938454
14) Interval dot product using
    ivectors and Dot0 (accu):
Penalty   35.0,  time used  2.6678440571
```

These results show for example that the calculations of a real scalar product with the help of the long accumulator (parameter $K = 0$) are 7.4 times slower than the simple C++ loop on the selected machine (result 6). If one simulates quadruple precision (double-double accuracy, parameter $K = 2$) with the help of the DotK algorithm, it is just 2.3 times slower than the simple C++ loop and thus more than 3 times faster than the calculation with the long accumulator.

Here it should be emphasized again that the usage of a long accumulator realized in hardware would lead to identical working speed as the simple C++ loop. Unfortunately, this hardware support is not available on current processors. The scalar product via the long accumulator would not only be always maximally accurate (this also applies to the software solutions), but also about 2.3 times faster than the DotK algorithm with $K = 2$. Since the DotK algorithm for $K > 1$ always needs more operations than the simple C-loop, this algorithm stays slower even with appropriate hardware support and thus also slower than the always exact scalar product using a hardware accumulator.

Michael Zimmer
Walter Krämer
Werner Hofschuster
Department of Mathematics and Computer Science
University of Wuppertal
Gaußstraße 20
D-42097 Wuppertal
e-mail: michael.zimmer@math.uni-wuppertal.de
e-mail: Walter.Kraemer@math.uni-wuppertal.de
e-mail: hofschuster@math.uni-wuppertal.de

Gerd Bohlender
Institute for Applied and Numerical Mathematics 2
KIT – Karlsruhe Institute of Technology
Kaiserstraße 12, D-76131 Karlsruhe
e-mail: gerd.bohlender@kit.edu

Received November 9, 2009
Final Accepted May 20, 2010