Efficient Matrix Multiplication Based on Discrete Stochastic Arithmetic*

Sethy Montan and Christophe Denis
EDF R&D - Département SINETICS - 1, Avenue du général de Gaulle 92141 Clamart Cedex - France
christophe.denis@edf.fr,sethy.montan@edf.fr

Jean-Marie Chesneaux and Jean-Luc Lamotte
Laboratoire d'Informatique de Paris 6 - Université Pierre et Marie Curie, 4 place jussieu 75005 - France
jean-marie.chesneaux@lip6.fr,jean-luc.lamotte@lip6.fr

Abstract

Numerical verification of industrial codes, such as those developed at Électricit e de France (EDF) R&D, requires estimating the precision and the quality of computed results, which is even more challenging for codes running in HPC environments where billions of instructions are performed each second, usually using external libraries (e.g., MPI, BLACS, BLAS, LAPACK). In this context, one needs a tool that is as nonintrusive as possible to avoid rewriting the original code. In this regard, the CADNA library, which implements the Discrete Stochastic Arithmetic, appears to be a promising approach for industrial applications.

In this paper, we are interested in an efficient implementation of the BLAS routine DGEMM (General Matrix Multiply) using Discrete Stochastic Arithmetic. The implementation of the basic algorithm for a matrix product using stochastic types leads to an overhead greater than 1000 for a matrix of 1024*1024 compared to the standard version and commercial versions of xGEMM. We present details of different solutions to reduce this overhead and results we have obtained.

Keywords: Linear Algebra, Matrix Multiply, Round-off Error, Numerical verification, Discrete Stochastic Arithmetic

AMS subject classifications: 65G20, 65G40, 65G50, 65Fxx

1 Motivation

Several sources of errors and approximations occur during any numerical simulation: physical phenomena are observed with measure errors and modelled using mathematical equations, continuous functions are replaced by discretized ones and real numbers

*Submitted: February 10, 2013; Revised: October 28, 2014; Accepted: November 7, 2014.

are replaced by finite-precision representations (floating-point numbers). IEEE-754 arithmetic generates round-off errors at each elementary arithmetic operation. These errors can accumulate to affect the accuracy of computed results, possibly leading to partial or total inaccuracy.

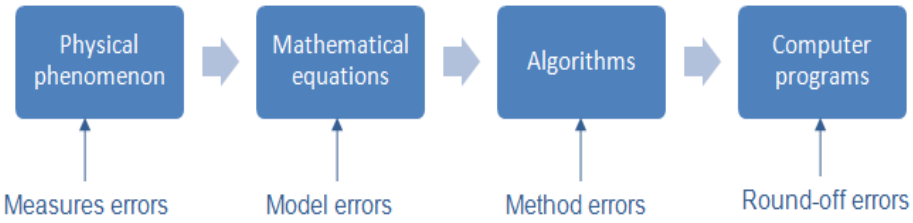


Figure 1: Numerical simulation process

Numerical verification focuses on round-off error propagation. It is crucial, especially for industries where it is required to estimate the precision and the quality of the computed results. It is even more important now when most codes are run in High Performance Computing environments where billions of instructions are performed per second. The effect of rounding errors can be analysed by multiple methods including forward/backward analysis, interval arithmetic or discrete stochastic arithmetic.

In this paper, we focus on the numerical verification of industrial programs. In this context, practitioners need a tool that is as non-intrusive as possible to avoid rewriting the original code. In this regard, the CADNA library [15], which implements discrete stochastic arithmetic, appears to be a promising approach. However, to improve performance, industrial programs usually use external libraries (e.g., MPI, BLACS, BLAS, LAPACK) [4]. Theses libraries are highly optimized to obtain good computational performance, nearly peak performance in some cases. CADNA provides new numerical types, called stochastic types, on which round-off errors can be estimated. These new types are not compatible with the aforementioned libraries. Therefore, it is necessary to develop extensions for these external libraries to perform a total and efficient numerical verification on industrial programs.

We consider an efficient implementation of the BLAS (Basic Linear Algebra Subprograms) Level 3 routines compatible with CADNA. We focused on the *xGEMM - General Matrix Multiply* routine, calling our new routine *DgemmCADNA*. For BLAS Level 1 and 2 subprograms, sufficient performance can be achieved by using a template version of BLAS [23]. The implementation of a basic algorithm for a matrix product compatible with stochastic types leads to an overhead greater than 1000 for a matrix of 1024×1024 compared to the standard and commercial versions of xGEMM. This overhead is due to the use of stochastic types, whose rounding mode changes randomly at each elementary operation ($\times$, $/$, $+$, $-$) and a non-optimized use of memory.

Outline of the paper. In section 2, we present the main numerical validation tools and especially the CADNA library (section 2.3). After a brief presentation of BLAS routines (section 3.1), we present the problem of getting an efficient matrix multiplication routine compatible with CADNA Library (section 3.2) and its implementation (section 4). We compare our routine with other accurate matrix multiplication subroutines. Finally, we present the main results in section 5.

2 Numerical Validation

2.1 Rounding Errors

Rounding errors present an inherent problem to all computer programs in which numbers are represented in a finite form: a sequence of symbols (0 and 1 in base 2). D. Goldberg has pointed out the importance of rounding error propagation in numerical programs with his article "*What every computer scientist should know about floating-point arithmetic*" [8]. An excellent overview on rounding error also can be found in [12, 18]. A numerical program is a sequence of arithmetic operations, where error can occur at every operation, potentially leading to a loss of accuracy. Since it is intrinsically impossible to avoid rounding errors, we try to control their propagation by analysing the errors and by providing a bound on the error of computed results or by trying to improve the accuracy of results.

2.2 Numerical Verification in an Industrial Context

In an industrial context, numerical verification can be performed in two steps. The first step analyses a code or an algorithm (e.g., using forward/backward analysis, interval arithmetic or discrete stochastic arithmetic) to identify any potential numerical instabilities and parts of the code which generate these instabilities. The second step finds methods and tools to improve the accuracy of the code.

The goal of forward analysis is to estimate or bound the distance between the exact solution y and the computed solution $\widehat{y}$, which is called forward error. Backward analysis computes the distance between the initial problem and a problem that is solved exactly. The computed solution $\widehat{y}$ is assumed to be the exact solution, and one seeks the backward error Δ_x for which $\widehat{y} = f(x + \Delta_x)$.

The principle of interval arithmetic is to enclose every number in an enclosing interval (a real number x is represented by an interval $[\underline{x}, \overline{x}]$). Results of operations are intervals covering the range of all possible outcomes. Interval arithmetic offers guaranteed bounds for each computed result. The main drawback is the necessity to rewrite codes, often using specialized algorithms to avoid interval overestimation.

Discrete stochastic arithmetic [2] is based on the CESTAC method [24], which is described briefly in section 2.3 and implemented in the CADNA library.

Improving the numerical accuracy of a code can be achieved by increasing the initial precision of floating numbers as done in the Multiple Precision Floating-Point Reliable library - MPFR.[1] Based on GNU Multi-Precision library, MPFR is a portable C library for arbitrary-precision binary floating-point computation with correct rounding [7]. Use of the MPFR library does not guarantee the accuracy of the summation, unlike the Multiple Precision Floating-point Interval library[2] (MPFI) [21], which combines interval arithmetic and multiple precision.

Another way to improve the accuracy of computed results is to use compensated algorithms. These algorithms estimate the rounding error and add it to the computed result with an error-free transformation (see equation 1). Consider the context of IEEE 754 floating-point arithmetic with rounding to nearest. If a and b are two floating numbers, the rounding error which occurs during $fl(a + b)$ is a floating number:

$$x + \delta = a + b, \quad \text{with} \quad x = fl(a + b) \quad \text{and} \quad \delta \quad \text{a floating number.} \tag{1}$$

[1] see http://www.mpfr.org
[2] see http://mpfi.gforge.inria.fr

An Error Free Transformation (EFT) can be generalised to multiplication and in another way to division. The floating point numbers x and δ can be computed easily and exactly with working precision. An excellent overview of EFT and its applications can be found in [19].

With any of these approaches, the cost of the numerical verification process is very important. Especially for industry, a significant cost for development or for execution implies additional financial costs for the company. Therefore, it is important to find the best compromise between costs and gains in accuracy. As a consequence, avoiding rewriting codes should be considered as a priority. In this regard, the numerical verification should be performed by a tool which is as inobtrusive as possible. The CADNA library appears to be one of the most promising candidates for numerical verification of industrial applications. Inserting it into a code is straightforward compared with other validation tools. In the next section (2.3), more details on Discrete Stochastic Arithmetic are given, and its implementation in the CADNA library is described.

2.3 The CADNA Library

The CADNA[3] library uses a probabilistic approach to estimate round-off error propagation by in any simulation program written in C/C++ or Fortran and to control its numerical quality by detecting numerical instabilities that may occur at run time [15].

The CESTAC[4] method runs the same code several times synchronously with a random rounding mode for each operation. Each run generates a different rounding propagation. The random rounding mode consists of rounding r towards $+\infty$ or towards $-\infty$ with probability 0.5. When the same code is executed N times, if round-off errors affect the result, even slightly, N different results are obtained from N different runs. A statistical test may be applied on these N samples. It has been proved [24] that each of the N samples (results) R_i can be modelled to the first order in 2^{-p} by a random variable R as

$$R \approx r + \sum_{i=1}^{n} u_i(d) 2^{-p} \alpha_i \ , \tag{2}$$

where r is the exact result, p is the number of bits in the mantissa, α_i are independent uniformly distributed random variables on $[-1, 1]$ and u_i are coefficients depending exclusively on the data and on the code.

The round-off error of the final floating-point result is estimated from the different computed results $R_i, i = 1, \ldots, N$. The mean value $\overline{R}$ of R_i is chosen as the computed result. The number of exact decimal significant digits C_R of $\overline{R}$ is estimated as

$$C_R = \log_{10} \left(\frac{\sqrt{N} \cdot |\overline{R}|}{\sigma \tau_\beta} \right) \ , \tag{3}$$

where

$$\overline{R} = \frac{1}{N} \sum_{i=1}^{N} R_i \ , \quad \sigma^2 = \frac{1}{N-1} \sum_{i=1}^{N} \left(R_i - \overline{R} \right)^2$$

and τ_β is the value of Student's distribution for $N - 1$ degrees of freedom and a probability level $1 - \beta$.

[3]The CADNA library is available for download at http://www-pequan.lip6.fr/cadna/

[4]The CESTAC method (Contrôle et Estimation Stochastique des Arrondis de Calculs) was propoed by M. La Porte and J. Vignes in 1974.

In practice, validation using the CESTAC method requires a dynamic control of multiplications and divisions during the execution of the code. This leads to the synchronous implementation of CESTAC (i.e., the parallel computing of the N samples R_i) and the concept of computational zero [15]. The classical float is replaced by a 3-sample $X = (X_1, X_2, X_3)$. Every elementary operation $\Omega \in (+, -, \times, /)$ is defined by $X\Omega Y = (X_1 \omega Y_1, X_2 \omega Y_2, X_3 \omega Y_3)$, where ω is the floating-point operation followed by a random rounding.

Definition 2.1 *During the run of a code using the CESTAC method, an intermediate or a final result R is a computational zero, also called an informatical zero, denoted by @.0, if $C_R \leq 0$ or $\forall i, R_i = 0$.*

Definition 2.2 *X is stochastically strictly greater than Y if and only if*

$$\overline{X} > \overline{Y} \quad \textit{and} \quad X - Y \neq @.0 \ .$$

.

Definition 2.3 *X is stochastically strictly greater than or equal to Y if and only if*

$$\overline{X} \geq \overline{Y} \quad \textit{or} \quad X - Y = @.0 \ .$$

.

The CESTAC method, combined with these new definitions, defines *Discrete Stochastic Arithmetic (DSA)*. The elements of DSA, named stochastic numbers, are N-sets provided by the CESTAC method. CADNA contains the definition of all arithmetic operations and order relations for the stochastic types. When a stochastic variable is printed, only its exact significant digits appear. For a computational zero, the symbol "@.0" is printed. More precisely, during the execution of any code, the library estimates the inaccuracy due to rounding error propagation to detect numerical instabilities, to check the sequencing of the program (tests and branching) and to estimate the accuracy of all the intermediate computations.

3 The DgemmCADNA routine

3.1 The Basic Linear Algebra Subprograms

The BLAS routines provide standard building blocks for performing linear algebra operations. The routines are divided into three levels: Level 1 for operations on vectors (ex., xAXPY), Level 2 for matrix-vector operations (ex., xGEMV) and Level 3 for matrix-matrix operations (ex., xGEMM). The *xGEMM* subprogram performs matrix multiplication, one of the most common numerical operations, especially in the area of dense linear algebra. It forms the core of many important algorithms, including linear systems solvers, least square problems and singular and eigenvalue computation.

There are many BLAS implementations. The Netlib BLAS [17] is the reference implementation. Other implementations often are optimised for a given architecture. Well-known implementations include ATLAS (*Automatically Tuned Linear Algebra Software*) [3], which automatically generates an optimised version adapted to the architecture on which it is installed, GotoBLAS [10, 9] and Intel MKL [13]. Table 1 gives the performance (Gflops) of different BLAS implementations for xAXPY (Level

1), xGEMV (Level 2) and xGEMM (Level 3) for a 4096×4096 matrix and a 4096-vector. The computer used for all tests is described in Table 3. The two most highly optimised versions (GotoBLAS and Intel MKL) achieve the best performance, especially for xGEMM (very close to the machine peak performance). For BLAS levels 1 and 2, performance of the Netlib and the other implementations are close. In general, only Level 3 subprograms fully exploit the machine characteristics, for reasons given in [3]. Due to these performance statistics and its importance, we chose to focus on the DGEMM subprogram.

Table 1: BLAS implementation performance (Gflops) for axpy, gemv and gemm

	Single Precision			Double Precision		
versions	saxpy	sgemv	sgemm	daxpy	dgemv	dgemm
Netlib	1.18482	1.24672	2.6391	1.18482	1.15347	1.35378
Atlas	1.18482	1.82857	3.28395	0.928642	1.50138	5.55025
Mkl 1 threads	6.87195	4.2074	15.1008	2.02116	2.11232	7.53686
Goto 1 threads	8.58993	4.46928	15.38	2.86331	2.12331	7.52166
Mkl 8 threads	2.02116	5.66508	112.89	1.63618	2.79974	58.0523
Goto 8 threads	1.37439	8.95505	115.444	1.63618	4.60287	56.3343

3.2 Direct Implementation of DSA in xGEMM

The objective of this implementation is to identify the possible overhead due to the use of stochastic types. The direct implementation is a basic algorithm with three inner loops (See Listing 1). This first version, named *dgemmcadnaV1*, has been compared to *dgemmcadnaV2*, which is the same code, but the calls to random rounding modes are removed. The difference between the two versions is presented in Table 2. *dgemmcadnaV1* is compared to the other BLAS implementations and to LinAlg [23], which is a template (C++) implementation of Netlib BLAS. In our experiment, templates are replaced by stochastic types.

Listing 1: Direct implementation of DSA in xGEMM

```
int dgemmcadnaV1(int n, double_st alpha, double_st *A, ↩
    double_st *B, double_st beta,double_st *C){
  int i, j, k;
  for (i = 0; i < n; i++){
    for (j = 0; j < n; j++){
      for (k = 0; k < n; k++){
        C[i*n+j] += alpha* A[i*n+k] * B[k*n+j] ;
      } /* for k */
    } /* for j */
  } /* for i */
}
```

These two experiments point out the poor performance of the *dgemmcadnaV1* routine compare to the others. In fact, the time ratio $V1/V2$ is greater than 7. This important overhead is the direct consequence of the many rounding mode changes. More than 85% of the execution time of $V1$ is due to random rounding mode selection (see Table 2). Indeed, a system function is used to change the rounding mode,

Table 2: Overhead due to the discrete stochastic arithmetic: computed measured time of DgemmCadnaV1 is compared to DgemmCadnaV2; the random rounding function calls have been removed in DgemmCadnaV2

Size	*DgemmCadnaV1*	*DgemmCadnaV2*
512	34.728	2.247
1024	320.174	40.660
2048	2636.270	372.290

and it automatically breaks the instructions pipeline. In the worst case, the processor executes only one instruction per cycle. Besides, the worst performance of *dgemmcadnaV1* and LinAlg compared to Netlib is due to the use of DSA and the three inner loops. Using DSA, each arithmetic operation is done three times, using three times more floating point numbers and four times more memory. The loops cause cache and TLB misses. Note that other versions of DGEMM were highly optimized to exploit fully the performance offered by the machine.

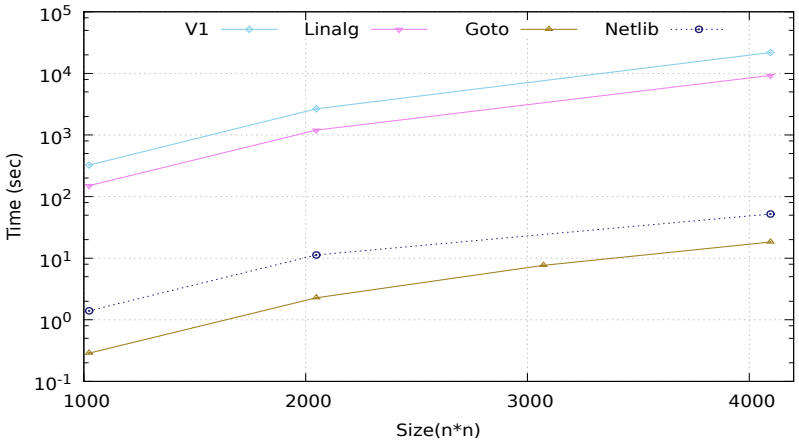


Figure 2: DgemmCadnaV1 compared to other implementations of BLAS (LinAlg, GotoBLAS, Netlib). GotoBLAS is 1000× faster than DgemmCadnaV1

4 Optimisation of DgemmCADNA

Implementation of a matrix product with optimum performance is a very complex problem. Kazushige Goto has written in [9]:

"Implementing matrix multiplication so that near-optimal performance is attained requires a thorough understanding of how the operation must be layered at the macro level in combination with careful engineering of high-performance kernels at the micro

level."

The most important parameter is the block size. Using blocks (tiles) allows a better use of memory. We can exploit the characteristics of actual machines which are based on the principle of shared memory with NUMA (Non-Uniform Memory Access). These machines consist of several processors, each containing multiple cores. Each core is associated with a memory unit, and they are interconnected by hierarchical cache memory, giving them transparent access to the entire memory [5, 6]. These architectures have a very hierarchical structure on which the data access depends on their location in the memory. The block size must be chosen so that all sub-matrices involved in the computation fit into the targeted memory area.

Besides the hierarchical structure, the *Translation Look-aside Buffer* (TLB) stores the addresses of the data used most recently to accelerate the translation of virtual address to physical address. The most significant difference between a cache miss and a TLB miss is that a cache miss does not necessarily stall the CPU.

Many studies have been devoted to the optimization of the computation of matrix products on various architectures (CPU, GPU, CPU CELL). For example, the emerging trend in linear algebra is the use of specialized data structures such as *Block Data Layout* (BDL) [20] and expression of algorithms directly in terms of kernels [16].

In this paper, our purpose is slightly different from that of the previous work. We develop an efficient matrix multiplication algorithm adapted to the datatypes used in the CADNA library to limit its overhead. The floating point datatypes in the classical matrix multiplication are replaced with CADNA stochastic datatypes composed of

- three double precision floating point numbers and one integer to replace a double precision floating point datatype; and
- three single precision floating point numbers and one integer to replace a single precision floating point datatype.

We were largely inspired by [1, 3, 9, 10, 11, 16, 22]. The main motive of all these implementations was to ***Reduce the number and the cost of memory access*** to reduce TLB and cache misses. The main solution is to use tiled algorithms, optimize cache locality and exploit temporal and spatial locality. We present the different steps for the optimisation in the next sections.

4.1 Exploiting Temporal and Spatial Locality

4.1.1 Iterative Tiled Algorithms

To reduce cache and TLB misses, it is very important to reduce data transfers. The best solution is to use the data as much as we can and as soon as we get it (temporal locality). A tiled algorithm exploits the spatial locality. Matrices are subdivided into sub-matrices $(C_{i,j}, A_{i,j}, B_{i,j})$, and the computations are made by block.

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1N} \\ C_{21} & C_{22} & \dots & C_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ C_{N1} & C_{N2} & \dots & C_{NN} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1N} \\ B_{21} & B_{22} & \dots & B_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ B_{N1} & B_{N2} & \dots & B_{NN} \end{bmatrix},$$

where every block C_{ij} of matrix C is computed by

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \ .$$

Listing 2 shows an implementation for the iterative implementation.

Listing 2: Iterative Tiled algorithm implementation

```
int SIZEBLOCK = ....;
int i,j,k,ii,jj,kk ;

for (i = 0; i < n/SIZEBLOCK; i++)
 for (j = 0; j < n/SIZEBLOCK; j++)
  for (k = 0; k < n/SIZEBLOCK; k++)
   for (ii = 0; ii < SIZEBLOCK; ii++)
    for (jj = 0; jj < SIZEBLOCK; jj++)
     for (kk = 0; kk < SIZEBLOCK; kk++)

        C[(i*n + j)*SIZEBLOCK +ii*n + jj] += ↩
            A[(i*n+k)*SIZEBLOCK + ii*n +kk] + ↩
            B[(k*n+j)*SIZEBLOCK + kk*n + jj] ;
```

On current machines, the cache is divided into three levels: L1, L2 and L3. Differences between the three levels of cache and different interactions with the CPU are explained in [5]. The L1 cache is the smallest and least expensive, and the L3 the most expensive and the largest in terms of memory size. In our case, we need to store three stochastic sub-matrices so that they can fit in the L1 cache (64 KB). However, half of the L1 cache is reserved for machine instructions [14].

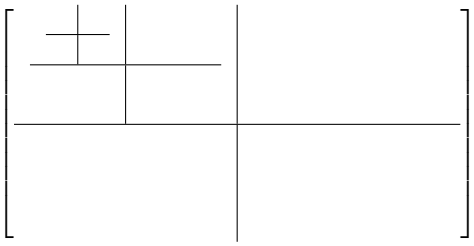
With the block size, $SIZEBLOCK$, the size of a *double_st*, $SizeDst$, and the available size of L1 cache, $SizeL1$, equation 4 must be verified. It holds for $SIZEBLOCK = 18$, but $SIZEBLOCK = 16$ is better, avoiding memory alignment problems.

$$3 \times SIZEBLOCK \times SIZEBLOCK \times SizeDst \leq SizeL1 \ . \tag{4}$$

4.1.2 Recursive Tiled Algorithms

The easiest way to exploit temporal locality is recursively [11], which exploits of all levels of cache memory. A recursive algorithm subdivides matrices into four submatrices and repeats the operation until it obtain blocks which can fit in the L1 cache.

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$



At each level of recursion, the partial results of blocks C are

$$\begin{aligned} C_{11} &= A_{11} \times B_{11} + A_{12} \times B_{21} \\ C_{12} &= A_{11} \times B_{12} + A_{12} \times B_{22} \\ C_{21} &= A_{21} \times B_{11} + A_{22} \times B_{21} \\ C_{22} &= A_{21} \times B_{12} + A_{22} \times B_{22} \ . \end{aligned}$$

4.1.3 An Iterative Tiled Algorithm Based on the Hardware

The idea here is to adapt the partitioning to fit the hierarchical memory: three levels of partitioning, one level for every cache level. The matrix (sub-matrices) are partitioned into sub-matrices. At each step, three blocks must fit in this level of cache memory.

1. First level for Cache L3: $A(n \times n)$ is divided into sub-matrices $A_{i,j}$

$$A(n \times n) = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix}$$

2. Second level for Cache L2: $A_{i,j}$ is divided into sub-matrices $AA_{i,j}$

$$A_{i,j} = \begin{bmatrix} AA_{11} & AA_{12} & \dots & AA_{1K} \\ AA_{21} & AA_{22} & \dots & AA_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ AA_{K1} & AA_{K2} & \dots & AA_{KK} \end{bmatrix}$$

3. Third level for Cache L1: $AA_{i,j}$ is divided in to sub-matrices $AAA_{i,j}$

$$AA_{ii,jj} = \begin{bmatrix} AAA_{11} & AAA_{12} & \dots & AAA_{1P} \\ AAA_{21} & AAA_{22} & \dots & AAA_{2P} \\ \vdots & \vdots & \ddots & \vdots \\ AAA_{P1} & AAA_{P2} & \dots & AAA_{PP} \end{bmatrix}$$

Block sizes at each level are determined from the characteristics of our machine (See Table 3). Our goal is that the data processed in L1 can use the locality of the data in the L2 cache and those processed at L2 and L3. We have blocks of size 128 for the first level (L3), 32 for level L2 and 16 for the last level (L1); these sizes were calculated from equation 4.

4.1.4 Using the Block Data Layout

The use of blocks maximizes the benefit of temporal locality of data for a given cache size. The previous solutions are designed to minimize cache misses by reducing the size of the matrices involved simultaneously in computation. There are also other techniques to reduce cache misses such as using *padding*. However, these techniques do not have a strong influence on TLB performance. Once the matrix sizes become larger, TLB performance becomes more important. The more TLB faults there are, the more the overall performance is degraded. Therefore, to optimize a given application, it has been proposed [20] to modify matrix storage models and the inner loops.

Traditionally, matrices are stored either in column or row major order:

- *Column Major order*

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 1 & 4 & 7 & 2 & 5 & 8 & 3 & 6 & 9 \end{bmatrix}$$

- *Row Major order*

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

However, when performing a matrix multiplication, the elements of matrix A are accessed in row major order, and the elements of B are accessed in column major order. If blocks are used, we need to make a big jump in memory to move from one row to another or from one column to another. We have the same problem when we pass from one block to another. As a result of jumps in memory, cache and TLB misses increase. The BDL defines the data storage model. The key idea of this approach is to reorganize the layout of matrix data stored in the main memory to make it cache friendly, and that the data layout matches the data access pattern.

Consider an $n \times n$ matrix A partitioned into $N \times N$ submatrices A_{ij}:

$$A(n \times n) = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix} \quad A_{ij}(p \times p) = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p1} & a_{p2} & \dots & a_{pp} \end{bmatrix}.$$

Data within one such block A_{ij} are mapped onto contiguous memory:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} & a_{21} & a_{22} & \dots & a_{2p} & \dots & a_{p1} & a_{p2} & \dots & a_{pp} \end{bmatrix},$$

and theses blocks are arranged in *row-major order*:

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} & A_{21} & A_{22} & \dots & A_{2N} & \dots & A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix}.$$

This storage model can be called *Block Row Major Layout (BRML)*. Using BDL can significantly improve performance and minimize TLB and cache misses on hierarchical memory machines. It involves copying and reorganization of matrices. As matrix B is accessed by column, the elements of the blocks are stored in *column-major* order. The matrix A is stored in BRLM format.

$$A(4\times4) = \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right] \Rightarrow A'(4\times4) = [1\,2\,5\,6\,3\,4\,7\,8\,9\,10\,13\,14\,11\,12\,15\,16]$$

$$B(4\times4) = \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right] \Rightarrow B'(4\times4) = [1\,5\,2\,6\,3\,7\,4\,8\,9\,13\,10\,14\,11\,15\,12\,16]$$

4.2 Reduce the Overhead Due to DSA

4.2.1 New Implementation of the CESTAC Method

In section 2.3, we presented Discrete Stochastic Arithmetic and the random rounding mode. In practice, the addition of two stochastic numbers a and b is done as

```
C[i].x = A[i].x + B[i].x ;
if (random) rnd_switch();
C[i].y = A[i].y + B[i].y ;
if (random) rnd_switch();
C[i].z = A[i].z + B[i].z ;
rnd_switch();
```

The four rounding modes of the IEEE-754 standard can be reduced to two rounding modes: towards $+\infty$ and towards $-\infty$ (see section 2.3). Consequently, we can consider that a basic mathematical operation between two stochastic data uses only two different rounding modes. Therefore, if we consider an operation between two stochastic vectors, we can group operations into two parts and change the rounding mode only twice, one for each part of the operation. For example, the addition of two stochastic vectors of four elements $C_i = A_i + B_i;\ 0 \leq i \leq 4$ could be implemented as shown in Listing 3. This new implementation improves exploitation of the pipeline length. It is now possible to execute more than one operation before a call to the rounding mode changing function *rnd _Switch()*. The idea is to do the maximum of operations to fully exploit the pipeline. Finally, two rounding mode changes are made instead of 12 in the original implementation.

Listing 3: New implementation for CESTAC

```
if(random) rnd_switch()
C[i].x = A[i].x + B[i].x ;
C[i].z = A[i].z + B[i].z ;
C[i+1].z = A[i+1].z + B[i+1].z ;
C[i+2].x = A[i+2].x + B[i+2].x ;
C[i+2].y = A[i+2].y + B[i+2].y ;
C[i+3].x = A[i+3].x + B[i+3].x ;
rnd_switch();
C[i].y = A[i].y + B[i].y ;
C[i+1].x = A[i+1].x + B[i+1].x ;
C[i+1].y = A[i+1].y + B[i+1].y ;
C[i+2].z = A[i+2].z + B[i+2].z ;
C[i+3].y = A[i+3].y + B[i+3].y ;
C[i+3].z = A[i+3].z + B[i+3].z ;
```

4.2.2 On the Validity of the New Implementation

To improve the performance of DgemmCADNA, we have proposed a new implementation for DSA. We need to be sure that we are still using the CESTAC method and following the principles of the method.

Few reminders on the CESTAC method. A computed result (a sequence of arithmetic operations) can be modelled by equation 2. More precisely, equation 2 is derived from equation 5,

$$R = r + \sum_{i=1}^{Sn} g_i(d) 2^{E_i - p} \varepsilon_i (\alpha_i - h_i) \ , \tag{5}$$

where g_i are constant values depending on the data and the algorithm, E_i are exponents of intermediary results, α_i are the part lost due to round-off error, h_i are random

perturbations, ε_i are intermediary results signs, r is the correct mathematical result and n is the number of operations during the execution.

Equation 2 (respectively, equation 5) has been established on the basis that two hypotheses hold: i) the round-off error α_i (respectively, $(\alpha_i - h_i)$) are independent, centered uniformly distributed random variables; and ii) the approximation to the first order in 2^{-p} is legitimate.

On the implementation. In the implementation of DSA, R is replaced by a 3-sample Rx, Ry, Rz. Consequently, equation 5 becomes

$$\begin{aligned} R_x &= r + \sum_{i=1}^{Sn} g_{x_i}(d) 2^{E_{x_i} - p} \varepsilon_i (\alpha_{x_i} - h_{x_i}) \\ R_y &= r + \sum_{i=1}^{Sn} g_{y_i}(d) 2^{E_{y_i} - p} \varepsilon_i (\alpha_{y_i} - h_{y_i}) \\ R_z &= r + \sum_{i=1}^{Sn} g_{z_i}(d) 2^{E_{z_i} - p} \varepsilon_i (\alpha_{z_i} - h_{z_i}) \ . \end{aligned} \tag{6}$$

In fact, $h_i \in \{-1; 1\}$; h_{x_i} and h_{y_i} are chosen randomly; $h_{z_i} = \overline{h_{y_i}}$. Choosing h_i means choosing a rounding mode.

On the new implementation. If we consider operations by groups of four, equation 5 is equivalent to

$$R = r + \sum_{k=1}^{Sn'} \sum_{j=1}^{4} p_{k_j} \ , \tag{7}$$

where $Sn' = Sn/4$ and $p_{k_j} = g_{k_j}(d) 2^{E_{k_j} - p} \varepsilon_{k_j} (\alpha_{k_j} - h_{k_j})$, and then

$$\begin{aligned} R_x &= r + \sum_{k=1}^{Sn'} \sum_{j=1}^{4} p_{k_{x_j}} \\ R_y &= r + \sum_{k=1}^{Sn'} \sum_{j=1}^{4} p_{k_{y_j}} \\ R_z &= r + \sum_{k=1}^{Sn'} \sum_{j=1}^{4} p_{k_{z_j}} \end{aligned} \tag{8}$$

with

$$\begin{aligned} p_{k_{x_0}} &= g_{k_{x_0}}(d) 2^{E_{k_{x_0}} - p} \varepsilon_{k_{x_0}} (\alpha_{k_{x_0}} - h_{k_{x_0}}) \\ p_{k_{y_0}} &= g_{k_{y_0}}(d) 2^{E_{k_{y_0}} - p} \varepsilon_{k_{y_0}} (\alpha_{k_{y_0}} - h_{k_{y_0}}) \\ p_{k_{z_0}} &= g_{k_{z_0}}(d) 2^{E_{k_{z_0}} - p} \varepsilon_{k_{z_0}} (\alpha_{k_{z_0}} - \overline{h_{k_{y_0}}}) \\ p_{k_{x_1}} &= g_{k_{x_1}}(d) 2^{E_{k_{x_1}} - p} \varepsilon_{k_{x_1}} (\alpha_{k_{x_1}} - h_{k_{x_1}}) \\ p_{k_{y_1}} &= g_{k_{y_1}}(d) 2^{E_{k_{y_1}} - p} \varepsilon_{k_{y_1}} (\alpha_{k_{y_1}} - h_{k_{y_1}}) \\ p_{k_{z_1}} &= g_{k_{z_1}}(d) 2^{E_{k_{z_1}} - p} \varepsilon_{k_{z_1}} (\alpha_{k_{z_1}} - \overline{h_{k_{y_1}}}) \\ p_{k_{x_2}} &= g_{k_{x_2}}(d) 2^{E_{k_{x_2}} - p} \varepsilon_{k_{x_2}} (\alpha_{k_{x_2}} - h_{k_{x_2}}) \\ p_{k_{y_2}} &= g_{k_{y_2}}(d) 2^{E_{k_{y_2}} - p} \varepsilon_{k_{y_2}} (\alpha_{k_{y_2}} - h_{k_{y_2}}) \end{aligned}$$

$$\begin{aligned}
p_{k_{z_2}} &= g_{k_{z_2}}(d)2^{E_{k_{z_2}}-p}\varepsilon_{k_{z_2}}(\alpha_{k_{z_2}} - \overline{h_{k_{y_2}}}) \\
p_{k_{x_3}} &= g_{k_{x_3}}(d)2^{E_{k_{x_3}}-p}\varepsilon_{k_{x_3}}(\alpha_{k_{x_3}} - h_{k_{x_3}}) \\
p_{k_{y_3}} &= g_{k_{y_3}}(d)2^{E_{k_{y_3}}-p}\varepsilon_{k_{y_3}}(\alpha_{k_{y_3}} - h_{k_{y_3}}) \\
p_{k_{z_3}} &= g_{k_{z_3}}(d)2^{E_{k_{z_3}}-p}\varepsilon_{k_{z_3}}(\alpha_{k_{z_3}} - \overline{h_{k_{y_3}}}) \ .
\end{aligned} \tag{9}$$

In equation 9, eight h_i have been chosen randomly, and four depend on the last rounding mode. It is important ihat at every step, there are at least two different h_i's. Therefore, with the new implementation, Equation 9 can be rewritten as

$$\begin{aligned}
p_{k_{x_0}} &= g_{k_{x_0}}(d)2^{E_{k_{x_0}}-p}\varepsilon_{k_{x_0}}(\alpha_{k_{x_0}} - h_1) \\
p_{k_{y_0}} &= g_{k_{y_0}}(d)2^{E_{k_{y_0}}-p}\varepsilon_{k_{y_0}}(\alpha_{k_{y_0}} - h_2) \\
p_{k_{z_0}} &= g_{k_{z_0}}(d)2^{E_{k_{z_0}}-p}\varepsilon_{k_{z_0}}(\alpha_{k_{z_0}} - h_1) \\
p_{k_{x_1}} &= g_{k_{x_1}}(d)2^{E_{k_{x_1}}-p}\varepsilon_{k_{x_1}}(\alpha_{k_{x_1}} - h_2) \\
p_{k_{y_1}} &= g_{k_{y_1}}(d)2^{E_{k_{y_1}}-p}\varepsilon_{k_{y_1}}(\alpha_{k_{y_1}} - h_2) \\
p_{k_{z_1}} &= g_{k_{z_1}}(d)2^{E_{k_{z_1}}-p}\varepsilon_{k_{z_1}}(\alpha_{k_{z_1}} - h_1) \\
p_{k_{x_2}} &= g_{k_{x_2}}(d)2^{E_{k_{x_2}}-p}\varepsilon_{k_{x_2}}(\alpha_{k_{x_2}} - h_1) \\
p_{k_{y_2}} &= g_{k_{y_2}}(d)2^{E_{k_{y_2}}-p}\varepsilon_{k_{y_2}}(\alpha_{k_{y_2}} - h_1) \\
p_{k_{z_2}} &= g_{k_{z_2}}(d)2^{E_{k_{z_2}}-p}\varepsilon_{k_{z_2}}(\alpha_{k_{z_2}} - h_2) \\
p_{k_{x_3}} &= g_{k_{x_3}}(d)2^{E_{k_{x_3}}-p}\varepsilon_{k_{x_3}}(\alpha_{k_{x_3}} - h_1) \\
p_{k_{y_3}} &= g_{k_{y_3}}(d)2^{E_{k_{y_3}}-p}\varepsilon_{k_{y_3}}(\alpha_{k_{y_3}} - h_2) \\
p_{k_{z_3}} &= g_{k_{z_3}}(d)2^{E_{k_{z_3}}-p}\varepsilon_{k_{z_3}}(\alpha_{k_{z_3}} - h_2) \ ,
\end{aligned} \tag{10}$$

where $h2 = \overline{h1}$. Equation 10 is equivalent to equation 11:

$$\begin{aligned}
&1^{st}part \\
&p_{k_{x_0}} = g_{k_{x_0}}(d)2^{E_{k_{x_0}}-p}\varepsilon_{k_{x_0}}(\alpha_{k_{x_0}} - h_1) \\
&p_{k_{z_0}} = g_{k_{z_0}}(d)2^{E_{k_{z_0}}-p}\varepsilon_{k_{z_0}}(\alpha_{k_{z_0}} - h_1) \\
&p_{k_{z_1}} = g_{k_{z_1}}(d)2^{E_{k_{z_1}}-p}\varepsilon_{k_{z_1}}(\alpha_{k_{z_1}} - h_1) \\
&p_{k_{x_2}} = g_{k_{x_2}}(d)2^{E_{k_{x_2}}-p}\varepsilon_{k_{x_2}}(\alpha_{k_{x_2}} - h_1) \\
&p_{k_{y_2}} = g_{k_{y_2}}(d)2^{E_{k_{y_2}}-p}\varepsilon_{k_{y_2}}(\alpha_{k_{y_2}} - h_1) \\
&p_{k_{x_3}} = g_{k_{x_3}}(d)2^{E_{k_{x_3}}-p}\varepsilon_{k_{x_3}}(\alpha_{k_{x_3}} - h_1) \\
&2^{nd}part \\
&p_{k_{y_0}} = g_{k_{y_0}}(d)2^{E_{k_{y_0}}-p}\varepsilon_{k_{y_0}}(\alpha_{k_{y_0}} - \overline{h1}) \\
&p_{k_{x_1}} = g_{k_{x_1}}(d)2^{E_{k_{x_1}}-p}\varepsilon_{k_{x_1}}(\alpha_{k_{x_1}} - \overline{h1}) \\
&p_{k_{y_1}} = g_{k_{y_1}}(d)2^{E_{k_{y_1}}-p}\varepsilon_{k_{y_1}}(\alpha_{k_{y_1}} - \overline{h1}) \\
&p_{k_{z_2}} = g_{k_{z_2}}(d)2^{E_{k_{z_2}}-p}\varepsilon_{k_{z_2}}(\alpha_{k_{z_2}} - \overline{h1}) \\
&p_{k_{y_3}} = g_{k_{y_3}}(d)2^{E_{k_{y_3}}-p}\varepsilon_{k_{y_3}}(\alpha_{k_{y_3}} - \overline{h1}) \\
&p_{k_{z_3}} = g_{k_{z_3}}(d)2^{E_{k_{z_3}}-p}\varepsilon_{k_{z_3}}(\alpha_{k_{z_3}} - \overline{h1}) \ .
\end{aligned} \tag{11}$$

In this case, only $h1$ is chosen randomly. In the original implementation, for 12 operations, eight h_i are chosen randomly.

This new formulation affects the hypotheses on which equation 5 has been established, but only slightly. It is important that many random rounding modes were used in the computation. In a real life numerical simulation, even with the new implementation, they remain available. For example, if we consider the multiplication of two square 1024×1024 matrices, there are $2 \times 1024 \times 1024 \times 1024$ floating-point operations (2 Gflops). With DSA, there are 3×2 Gflops. With the first implementation, we have 4 Giga random h_i; with the new implementation, there are 0.5 Giga random h_i.

4.3 Kernel Optimisation

Matrix multiplication performance can be improved by reducing data transfers. Besides the macroscopic optimisation, microscopic optimisations can be done. Performance also can be improved by changing the loop order (inner and/or outer loop) and by optimizing the computation kernel (the inner operation to compute a partial result of block $C_{ij} = A_{ik} \times B_{kj}$). Indeed, these loops define how to access sub-matrices. With an adequate order, the number of memory access to read data (access to elements of A and B) can be reduced. Note that the number of memory access to write (access to element C) is constant. The inner loops are for the kernel, and the outer loops are loops on blocks. The number of branching tests (tests at the end of loops) can be reduced by unrolling loops. For example, consider Listing 4.

Listing 4: Inner loops

```
for(int i = 0; i < nb_block; i++){
  for(int k = 0; k < nb_block; k++){
    for(int j = 0; j < nb_block; j++){
      Cij = Aik * Bkj /*kernel*/
```

By unrolling these loops, we obtain

$$
\begin{aligned}
C_{11} &= A_{11} \times B_{11} \\
C_{12} &= A_{11} \times B_{12} \\
C_{11} &= A_{12} \times B_{21} \\
C_{12} &= A_{12} \times B_{22} \\
C_{21} &= A_{21} \times B_{11} \\
C_{22} &= A_{21} \times B_{12} \\
C_{21} &= A_{22} \times B_{21} \\
C_{22} &= A_{22} \times B_{22} \ .
\end{aligned}
$$

These optimisations access the elements of matrix A only once and in a favorable order. That is what is called "data re-use". In the next section, we will present the comparison of all the implementations and the main results.

5 Results

We will present in this section the performance of the previous solutions. Table 3 shows the characteristics of our test machine.

We have implemented all the optimisations proposed in section 4. As we explained, the optimum size for stochastic sub-matrices is 16. We compare the following implementations: B16 iterative tiled algorithm (section 4.1.1); DGBR16 recursive tiled

Table 3: Test machine characteristics.

Name	Processor	Cores	SIMD Date	GFlops th.
$_{2\times4}^{32}$Nehalem	2×Xeon E5504 2.00 GHz	2×4	SSE 4.2 03/2009	119.2

Memory	Cache		
	L1	L2	L3
4 Go DDR3 800 Mhz	4×64 Kio	4×256 Kio	4 Mio

algorithm (section 4.1.2) with block loops optimised and inner loops unrolled; DGBI16 adapted iterative tiled algorithm (section 4.1.3) with block loops optimised and the inner loop unrolled and BRML16 based on BDL (section 4.1.4), with block loops optimised and the inner loop unrolled. Figure 3 and Table 4 present our main results.

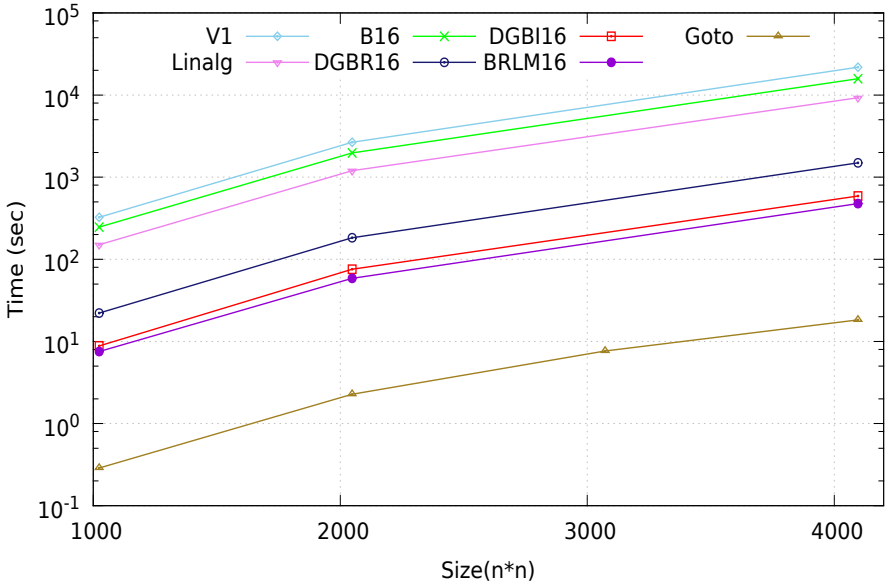


Figure 3: Different versions of DgemmCADNA compared to GotoBLAS and LinAlg.

The B16 version is better than DgemmCADNAV1, but the first good performance is obtained with DGBRI16, which is better than LinAlg. This can be explained by the fact that this implementation with different sizes of blocks is completely hardware dependent. However, the best version is the BRML16. Despite the copying and the reorganization of matrices, we obtain better results than the conventional block algorithms. These results confirm the importance of the data storage model.

Table 4: Comparision of DgemmCadnaV1, BRM16 and GotoBlas (one thread).

Taille	V1	BRML16	GotoBlas	V1/Goto	BRML16/Goto
1024	324.54	7.52	0.29	1127.40	26.12
2048	2658.72	58.69	2.27	1168.63	25.81
4096	21818.4	476.23	18.27	1194.06	26.06

Finally, all these optimisations have improved considerably the execution time. We obtained a gain of 45×, compared to the first version. Compared to GotoBlas, the primary overhead is about 1100, and now it is about 25. It is important to notice that our implementation needs three times more floating point operations and four times more memory due to the stochastic types.

We tried to improve the performance by using vector instructions SSE (Streaming SIMD Extensions) or AVX (Intel Advance Vector Extensions), but the performance is not encouraging. The use of vector instructions in a dot product is the easiest way to improve the execution time in double precision 2× with SSE and 4× with AVX. In the case of stochastic types, for a vector of size 10^6, we obtain a speed-up of 0.137575, which is obviously insufficient.

6 Conclusion

We present several candidates for an efficient implementation of matrix multiplication based on Discrete Stochastic Arithmetic. This arithmetic introduces an important overhead. Special data structures (Block Data Layout) are used to improve the matrix storage, and a new implementation of DSA has been introduced. This implementation reduces the overhead due to the random rounding mode of DSA. Finally, we have obtained an overhead about 25 compared to GotoBLAS in a sequential mode.

References

[1] Q. Bourgerie, P. Fortin, and J.L. Lamotte. Efficient complex matrix multiplication on the synergistic processing element of the cell processor. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–8, Heraklion, Crete, Greece, 2010. IEEE.

[2] J.M. Chesneaux. *L'arithmétique stochastique et le logiciel CADNA*. PhD thesis, Université Pierre et Marie Curie (UPMC), 1995. Habilitation à Diriger des Recherches.

[3] R. Clint Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[4] Christophe Denis and Sethy Montan. Numerical verification of industrial numerical codes. *ESAIM: Proc.*, 35:107–113, march 2012. `http://dx.doi.org/10.1051/proc/201235006`.

[5] U. Drepper. What every programmer should know about memory. 2007. `http://people.redhat.com/drepper/cpumemory.pdf`.

[6] M. Faverge. *Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-coeurs*. PhD thesis, LaBRI, Université Bordeaux I, Talence, France, December 2009. `http://www.labri.fr/~ramet/restricted/these_faverge.pdf`.

[7] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007. `http://doi.acm.org/10.1145/1236463.1236468`.

[8] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[9] Kazushige Goto and Robert van de Geijn. High performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1):4:1–4:14, July 2008. `http://doi.acm.org/10.1145/1377603.1377607`.

[10] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, May 2008. `http://doi.acm.org/10.1145/1356052.1356053`.

[11] P. Gottschling, D.S. Wise, and A. Joshi. Generic support of algorithmic and structural recursion for scientific computing 1. *International Journal of Parallel, Emergent and Distributed Systems*, 24(6):479–503, 2009.

[12] N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2002.

[13] Intel. Intel Math Kernel Library Reference Manual,Intel MKL 10.3 update 9. Technical report. `http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/index.htm`.

[14] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2011. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

[15] F. Jézéquel, J.M. Chesneaux, and J.L. Lamotte. A new version of the CADNA library for estimating round-off error propagation in Fortran programs. *Computer Physics Communications*, 181(11):1927–1928, 2010.

[16] J. Kurzak, W. Alvaro, and J. Dongarra. Optimizing matrix multiplication for a short-vector SIMD architecture-CELL processor. *Parallel Computing*, 35(3):138–150, 2009.

[17] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[18] J.M. Muller, N. Brisebarre, F. De Dinechin, C.P. Jeannerod, L. Vincent, and G. Melquiond. *Handbook of floating-point arithmetic*. Birkhauser, 2009.

[19] T. Ogita, S.M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[20] Neungsoo Park, Bo Hong, and Viktor K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14:640–654, 2003. `http://doi.ieeecomputersociety.org/10.1109/TPDS.2003.1214317`.

[21] N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11(4):275–290, 2005.

[22] G. W. Stewart. *Matrix Algorithms*. Society for Industrial and Applied Mathematics, Philadephia, PA, 1998. `http://dx.doi.org/10.1137/1.9781611971408`.

[23] Phillipe Trebuchet. The linalg library (lapack made generic). `http://www-apr.lip6.fr/~trebuche/linalg.html`.

[24] J. Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Numerical Algorithms*, 37(1):377–390, 2004.