Optimization of Triangular and Banded Matrix Operations Using 2d-Packed Layouts

Toufik Baroudi, Rachid Seghir, Vincent Loechner

Optimization of Triangular and Banded Matrix Operations Using 2d-Packed Layouts

TOUFIK BAROUDI, Department of Computer science, University of Batna 2
RACHID SEGHIR, LaSTIC Laboratory, University of Batna 2
VINCENT LOECHNER, ICube Laboratory, University of Strasbourg, and INRIA

Over the past few years, multicore systems have become more and more powerful, and thereby very useful in high-performance computing. However, many applications, such as some linear algebra algorithms, still cannot take full advantage of these systems. This is mainly due to the shortage of optimization techniques dealing with irregular control structures. In particular, the well-known polyhedral model fails to optimize loop nests whose bounds and/or array references are not affine functions. This is more likely to occur when handling sparse matrices in their packed formats. In this paper, we propose to use 2d-packed layouts and simple affine transformations to enable optimization of triangular and banded matrix operations. The benefit of our proposal is shown through an experimental study over a set of linear algebra benchmarks.

Additional Key Words and Phrases: Polyhedral Model, Code optimization and parallelization, Sparse matrices, 2d-packed layouts.

ACM Reference Format:
Toufik Baroudi, Rachid Seghir, and Vincent Loechner. 2017. Optimization of Triangular and Banded Matrix Operations Using 2d-Packed Layouts. *ACM Transactions on Architecture and Code Optimization* 14, 4, Article 55 (December 2017), 19 pages.
https://doi.org/10.1145/3162016

1 INTRODUCTION

Multicore systems have been drastically improved in the last few years. Therefore, they have gained a large ground in the high-performance computing world [7, 28, 32]. But in practice, many applications fail to take full advantage of the multicore architecture power. This issue can be addressed by using suitable automatic code transformation techniques in order to generate an optimized parallel code, without any effort from the programmer. Obviously, the new generated code has to fit, as much as possible, the architectural specificities of the target multicore system. One of the well-known approaches having this ability is the polyhedral model, which is born with the seminal work of Karp, Miller and Winograd on systems of uniform recurrence equations [26], and made widely applicable to static control programs by Feautrier [10, 21, 22]. Based on this powerful model, several optimization and parallelization techniques have been proposed since the early nineties.

Optimizing linear algebra programs is one of the issues that have attracted the attention of the code optimization research community for many years. Almost all the state-of-the-art optimization techniques target operations on dense matrix structures, in which array references and loop-index bounds are affine functions. However, operations on sparse matrices are the key computational kernels in many scientific and engineering applications [9, 15, 31, 37]. It is well known that dense matrix structures and algorithms are very inefficient when applied to sparse matrices, since they

use a large amount of memory to store zero elements and perform useless computations on them. Therefore, many alternative storage formats have been proposed in order to store and compute only non-zero elements. Among these formats, we can cite for example: (a) Linear Packed Format (LPF) where the matrix is stored in a single-dimensional array [18] used in the LINPACK Library [20] -the drawback of using this format is that the array references are no longer affine functions, and therefore the polyhedral model no longer applies on them; (b) Rectangular Full Packed Format (RFPF) [24] where a symmetric or triangular matrix is saved in rectangular format and alternative cholesky-factorization routines have been proposed in the LAPACK Library [4].

In this work, we propose a new approach to optimize triangular and banded matrix operations by using a *dense* 2-dimensional data structure for sparse-matrix storage. The basic idea is that the matrix operations using these data structures can be automatically optimized and parallelized by means of the polyhedral model. On one hand, triangular and banded matrix are compressed, which leads to significant savings in memory usage, and on another hand, the underlying code can be optimized and parallelized using existing polyhedral optimizing tools in order to achieve the best performance.

We demonstrate through experimental results the effectiveness of our approach by parallelizing and optimizing several matrix computation kernels using the state-of-the-art polyhedral compiler Pluto [12, 14], and comparing their performance to non-dense sequential and parallel versions, to the LPF version, and to the MKL library.

The remainder of the paper is organized as follows: in Section 2, a motivating example is given. Section 3 presents the state of the art: a background on the polyhedral model, some definitions on sparse matrices, and related work. Section 4 describes our proposed optimization technique for triangular and for banded matrices. Our experimental results are presented in Section 5. Finally, this work is concluded in Section 6.

2 MOTIVATING EXAMPLE

In this motivating example, we are interested in the automatic optimization and parallelization of the `sspfa` routine (see Figure 1) from the LINPACK Library [20]. The basic datatype is `double`. In this example, complex data dependences prevent standard compilers automatic optimization and parallelization, thus the need of a polyhedral compiler: complex loop transformations are required in order to expose tiling and coarse-grain parallelism.

When a matrix of order N is triangular, it is obviously not appropriate to store it in a structure of N^2 elements, since there will be $N(N-1)/2$ zero elements which are not required. Existing solutions usually utilize a linear packed layout, where the non-zero elements are stored in a single dimension array (a vector) of size $(N(N+1)/2)$. The corresponding `sspfa` code for such a structure is shown in Figure 2a.

One can notice that the array access functions in Figure 2a are not affine, and thereby not supported by the polyhedral model. Hence, static polyhedral compilers, such as Pluto [14], can not automatically optimize and parallelize matrix computations using this one-dimensional data structure to store non-zero elements. However, we would like to run a polyhedral compiler on this code, since complex data dependences prevent it from being easily parallelized, tiled, vectorized, and optimized for memory locality.

In our work, we have introduced a simple conversion from a square (unpacked) triangular matrix into a 2d-packed format, and affine transformations that fit the polyhedral model. Using our approach (as described below in Sect. 4), the `sspfa` routine is rewritten as shown in Figure 2b. In the transformed code all the array references, loop bounds and tests are affine, which means that this new code can be automatically optimized and parallelized by polyhedral compilers.

```
1        for (j = 0; j < n; j++)
2         for (k = 0; k < j; k++)
3          for (i = 0; i < k; i++)
4           A[j][k] -= A[j][i] * A[k][i];
5
```

Fig. 1. Code from the sspfa routine.

```
1   // moving A to AP
2   ii=0;
3   for(i=0; i< N; i++){
4    ii+=i;
5    for (j=0; j<i; j++){
6     AP[ii+j] = A[i][j];
7    }
8   }
9   // computing
10  jj=0;
11  for (j=0; j<N; j++){
12   kk=0;
13   jj+=j;
14   for (k=0; k<j; k++){
15    kk+=k;
16    for (i=0; i<k; i++){
17     AP[jj+k]-=AP[jj+i]*AP[kk+i];
18    }
19   }
20  }
```

(a) Linear Packed Format.

```
1   // moving A to A2
2   for(i=0; i< N; i++){
3    for (j=0; j<=i; j++){
4     if (2*i>N && 2*j>N)
5      A2[N-i-1][N-1] = A[i][j];
6     else
7      A2[i][j] = A[i][j];
8    }
9   }
10  // computing
11  for (j=0; j<N; j++){
12   for (k=0; k<j; k++){
13    for (i=0; i<k; i++){
14     if (2*i>N && 2*k>N)
15       A2[N-j-1][N-1]-=A2[N-j-1][N-i]
16                      *A2[N-k-1][N-i];
17     if (2*i<=N && 2*k>N)
18      A2[N-j-1][N-k]-=A2[j][i]*A2[k][i];
19     if (2*i<=N && 2*k<=N)
20       A2[j][k]-=A2[j][i]*A2[k][i];
21    }
22   }
23  }
```

(b) 2d-packed format.

Fig. 2. sspfa routine in linear and in 2d-packed formats.

To emphasize the effectiveness of the new 2d-packed version of the code, we have run Pluto on it (with options `--tile --parallel`), then we have compiled the resulting code using icc (version 18.0.0, with options `-O3 -march=native`) and executed it on a 20-cores Intel processor. For $N = 4000$, we measured that the execution time of the new version is about twelve times faster than the LPF version: it drops from 11.28 seconds to 0.95 seconds. The icc compiler could not auto-parallelize the LPF version -it has the same performance as the sequential one- neither could Pluto. The Pluto output from the 2d-packed code (not given here) is 140 lines long: the code was skewed, tiled (using the default tile size: $32 \times 32 \times 32$), and parallelized using OpenMP; no extra vectorization was detected by Pluto. The icc compiler could not auto-parallelize the original code, nor the 2d-packed code, so we need a polyhedral compiler to transform this code. Other measurements for this example are reported in the experimental section, column `sspfaTri` of Figure 12b.

3 STATE OF THE ART

3.1 Automatic Parallelization

Automatic parallelization is the process of automatically generating parallel codes from sequential algorithms [27, 29]. The resulting parallel code can be executed on parallel architectures without altering the semantics of the original code. The main advantage of such a process is that it does not require any effort of the programmer. The compiler takes as an input a sequential program and analyzes it in order to extract its dependences. Then, based on this information, the compiler generates a new optimized and parallel program that is semantically equivalent to the input program. This entire process is transparent from the programmer's point of view. But, of course, it requires lots of efforts from the compiler designers. More precisely, they usually deal with a formal mathematical model known as the polyhedral model.

3.2 The Polyhedral Model

Almost all scientific and engineering applications spend most of their execution time in small parts of their code, which are loop nests. These iterative structures have therefore attracted the attention of many researchers from the code optimization community, and have lead to the birth of the well-known polyhedral model [8, 21–23, 34]. The basic idea of this formalism is that each instance or iteration of a statement can be represented by an integer point within a convex region, called a *polyhedron*, defined by the statement enclosing loop bounds. These bounds have to be affine (linear functions with a constant part), and so have the data accesses, in order to fit the model. Once the different instances of all loop-nest statements are transformed into polyhedra, it is possible to compute inter and intra-statement dependences. This information is then used to produce legal program transformations that preserve the semantics of the original program. The new transformed code is more suitable and efficient for parallel execution and to better preserve spatial or temporal data locality. The polyhedral model may be viewed in terms of three phases: (a) static dependence analysis of the input program, (b) transformations in the polyhedral abstraction, and (c) generation of code for the transformed program.

3.3 Pluto

Pluto is a fully automatic polyhedral source-to-source transformation framework that can optimize regular programs for parallelism and data locality simultaneously [6, 12–14]. The basic idea of this framework is to transform an input C source code into a semantically equivalent output C code that achieves better parallelism and data locality. The targeted source codes are sequences of possibly imperfectly nested loops, and the transformations are affine functions that fit the polyhedral model. Pluto implements loop tiling, which is known by its good performance on large data arrays for both parallelism and locality. It also achieves SIMDization, by inserting compiler directives enabling vectorization of the inner loops. Other transformations include: scalar privatization, array contraction and many other loop transformations, such as loop fusion, reversal, interchange, skewing and unroll [12]. Furthermore, Pluto is able to automatically generate an optimized OpenMP parallel code for multicore architectures.

3.4 Triangular and Banded matrices

3.4.1 Triangular matrices. In the mathematical discipline of linear algebra, a triangular matrix is a special kind of square matrix. A square matrix is called lower triangular if all the entries above the main diagonal are zero. Similarly, a square matrix is called upper triangular if all the entries below the main diagonal are zero [2, 5]. There are two kinds of lower triangular matrices:

$$\begin{pmatrix} A_{00} & 0 & 0 & 0 & 0 \\ A_{10} & A_{11} & 0 & 0 & 0 \\ A_{20} & A_{21} & A_{22} & 0 & 0 \\ A_{30} & A_{31} & A_{32} & A_{33} & 0 \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \qquad \begin{pmatrix} A_{00} & 0 & 0 & 0 & 0 & 0 \\ A_{10} & A_{11} & 0 & 0 & 0 & 0 \\ A_{20} & A_{21} & A_{22} & 0 & 0 & 0 \\ A_{30} & A_{31} & A_{32} & A_{33} & 0 & 0 \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & 0 \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} \end{pmatrix}$$

Fig. 3. Example of even and odd triangular matrices.

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & 0 & 0 & 0 \\ A_{10} & A_{11} & A_{12} & A_{13} & 0 & 0 \\ 0 & A_{21} & A_{22} & A_{23} & A_{24} & 0 \\ 0 & 0 & A_{32} & A_{33} & A_{34} & A_{35} \\ 0 & 0 & 0 & A_{43} & A_{44} & A_{45} \\ 0 & 0 & 0 & 0 & A_{54} & A_{55} \end{pmatrix}$$

Fig. 4. Example of a banded matrix.

- *odd triangular matrix*: a matrix is said odd triangular if all the non-zero elements are below the diagonal of the matrix, and the order of the matrix is an odd number N ($N = 2k + 1$).
- *even triangular matrix*: in the case of an even triangular matrix, we have the same definition as before, but the order of the matrix is an even number N ($N = 2k$).

Figure 3 illustrates two instances of triangular matrices with even, respectively odd, orders.

3.4.2 Banded matrices. A banded matrix A is a matrix whose non-zero elements are located in a band centered along the principal diagonal. For such matrices only a small proportion of the N^2 elements are non-zeros. A square matrix A has lower bandwidth $W_l < N$ and upper bandwidth $W_u < N$ if W_l and W_u are the smallest integers such that:

$$A_{ij} = 0 \ \forall i > j + W_l, \text{ and } A_{ij} = 0 \ \forall j > i + W_u$$

respectively. The maximum number of non-zero elements in any row is $W = W_l + W_u + 1$ [2]. Figure 4 shows a banded matrix of order $N = 5$ with a lower bandwidth $W_l = 1$ and an upper bandwidth $W_u = 2$.

The triangular and banded matrices require to be handled in a special fashion because they contain a large number of useless zero elements. Indeed, there are $(N(N-1)/2)$ zeros in a triangular matrix of order N, and $((N - W_u)^2 + (N - W_l)^2 - 2N - W_u - W_l)/2$ zeros in a banded matrix of order N with W_u and W_l as upper and lower bandwidths, respectively. In particular, it is more efficient to store only non-zero elements in order to save memory space.

3.5 Other related work

Matrix computations are the cornerstone of numerous scientific and engineering applications, and the performances of almost all matrix-based applications are depending on those of matrix computations. Therefore, a huge interest of researchers has been directed towards matrix-computation optimization. However, most of the optimization research community has been targeting computations on *dense matrices* because of the regularity of their data storage layouts and underlying computations, in addition to their wide utilization [19]. Over the past three decades, many *manual* and *automatic* optimization techniques have been proposed.

Manual optimization techniques rely on the effort of the expert programmer, after a deep analysis of a given problem, to propose a manually tuned program whose performance is the best he can

achieve. Many manual matrix-computation optimizing techniques have been designed [16, 25, 35]. Alternatively, the automatic optimization techniques consist of designing and implementing tools and compilers that are able to translate a given program into an optimized and parallelized code. The programmer does not need to worry about the optimization process, all he needs to know is how to use these tools. Automatic optimizing techniques applied to matrix computations on dense matrices include Pluto [14], EPOD [17], ROSE [36], and PHiPAC [11].

The optimization techniques targeting matrix computations on *sparse matrices*, such as *triangular* and *banded* matrices, are rare and almost all of them are manual optimizing techniques applied to specific problems [1, 3, 30]. Among those works, Gustavson et al. [24] target Cholesky's algorithm on triangular matrices, by separating the sparse matrix computations in a set of computation on half-sized dense matrices, calling the Level 3 BLAS routines. Our representation format for triangular matrices is inspired by this work, where the matrix is cut into two parts, the small triangle being displaced to cover the zeros in the other part (as presented hereunder in Figure 7). We improved their storage format, in order to perform the same data transformation for both odd and even ordered matrices: in their proposal, odd and even ordered matrices are stored differently, which requires the programmer to distinguish between those two cases. Compared to their work, we also propose a new banded storage format.

Recently, Cui et al. [18] proposed an automatic layout-oblivious optimization technique for matrix computations. To the best of our knowledge, their work is the only one using a polyhedral compiler to optimize banded and triangular matrix computations. Their main idea is the isolation of the high-level semantics of operations from the organization details of compound data structures. This way, a simplified abstract specification of operations can be derived, and then accurately analyzed and optimized using the state-of-the-art optimizing compilers such as Pluto [14] and EPOD [17]. More precisely, their approach consists of three steps: *matrix normalization*, *optimization* and *matrix denormalization*. The fist step seeks to isolate the high-level semantics of matrix operations from the internal implementation of the data structures. This in turn consists of deriving a new abstract code handling unpacked data structures (dense matrices) from the original code handling packed data structures. In the second step, the resulting abstract code is fed to the source-to-source compiler to generate an optimized parallel code. This optimized code is then converted through the denormalization step in order to fit the original data structure organization. This mainly consists of converting back the two dimensional dense array accesses into one dimensional packed accesses. The authors propose an annotation language that the programmer has to use to define the intended semantics of data structures in their matrix computations, for performing steps 1 and 3 of their algorithm. Even if Cui et al.'s framework is well designed and proved to be effective in many cases, we believe that the programmers might have some difficulties when using their framework. On one hand, because they have to write the banded and triangular matrix computations in packed format (see Figure 2b for example), which they may not be familiar with. And on the other hand, because they have to express the intended semantics of data structures of their matrix computations in the annotation language, which may lead to errors. The authors indeed claim in their article that *the optimized code is guaranteed to be correct if the user-supplied annotations can be assumed to be correct*. They also mention that their approach is effective in optimizing only packed matrix computations where the matrix layout can be easily expressed with their annotation language. In our proposal, all the programmers have to do is to write the matrix computations in unpacked matrix format (two-dimensional arrays) and to declare them as triangular or banded. The entire optimization process is totally transparent and automatic from this point. In addition, Cui et al. point out that some layout-sensitive optimizations, *e.g.* SSE vectorization, need to be turned off at the optimization step and should be applied after the oblivious-layout optimization, while our

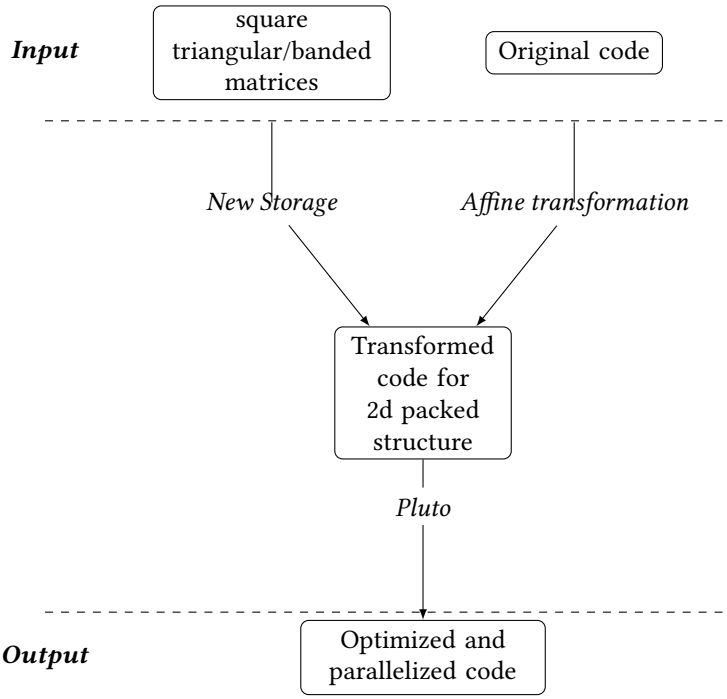


Fig. 5. Overview of the 2d-packed optimization technique.

proposal lets the optimizing polyhedral compiler take the right decisions for efficient data locality and vectorization.

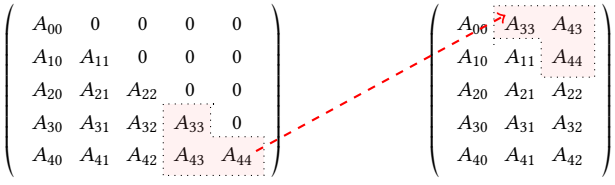
Very recently, Sampaio et al. [33] suggested to use a complex hybrid (static and dynamic) framework in order to overcome the limitations of purely static dependence-analysis techniques. Unlike our approach, in which we propose changing the data layout for specific cases so that the underlying codes are easy to analyze, Sampaio et al.'s work tries to optimize more general non-affine programs. But it is focusing in its current form, on demonstrating the feasibility and potential impact of using polyhedral transformations on non-affine programs. The evaluation of their approach for the polynomial access resulting from the compacted storage of triangular matrices is left to a future work.

4 THE 2D-PACKED FORMAT OPTIMIZATION TECHNIQUE

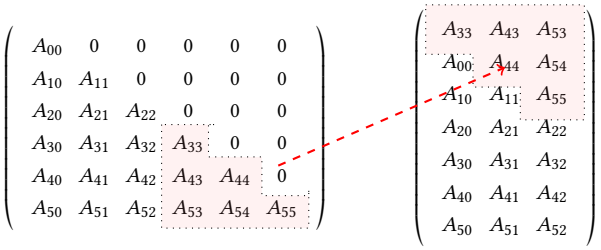
In this section, we describe our 2d-packed format optimization technique. It takes as input the sparse matrices to be handled and an original dense code. At first, our technique will define the sparse matrices in 2d-packed formats, where only non-zero elements are stored (2d-packed matrices are dynamic allocated). Then, the original code for dense matrices is transformed into a new code using the 2d-packed structures. The resulting code is finally parallelized and optimized by the Pluto source-to-source parallelizer and optimizer. An overview of our approach is shown in Figure 5.

4.1 2d-packed format optimization technique for triangular matrices

Gustavson et al. [24] propose to use rectangular full packed format to store a triangular matrix of order N. The size of the rectangle is $N \times (N+1)/2$ when N is odd, and $(N+1) \times N/2$ otherwise

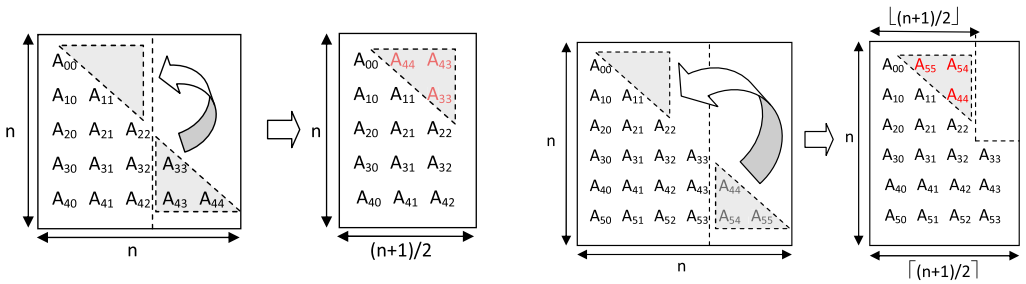


(a) Even triangular matrix.



(b) Odd triangular matrix.

Fig. 6. Storage in Rectangular Full-Packed Format.



(a) Odd triangular matrix.

(b) Even triangular matrix.

Fig. 7. Matrix storage in 2d-packed format.

(see Figure 6). Even if this representation guarantees to obtain a full rectangular structure in both cases, it has the disadvantage of adding an extra row in case N is even. This leads to write two versions of the code using two distinct transformation functions, depending on whether N is odd or even. Gustavson et al. also propose matrix transformations for the Cholesky factorization to fit the new data structure and their implementation in the LAPACK Library [4].

In our work, we propose to use the exact same data structure when N is odd. That is to say, a matrix of size $N \times (N+1)/2$ (N rows and $(N+1)/2$ columns) is required. When N is even, we propose a different data structure where a matrix of size $N \times N/2$ (N rows and $N/2$ columns) and an extra column of size $N/2$ are used to store the non-zeros values (see Figure 7). This new structure has the advantage to preserve the same code as in the odd case and only one transformation function is required. Algorithm 1 shows the way we store a triangular matrix in the 2d-packed format. In order to be able to use the new 2d-packed structure, we have proposed a piecewise unimodular transformation function, which one can use to transform any matrix operation handling triangular matrices. In our work, we have applied this transformation to many matrix computation benchmarks. The resulting codes are afterwards optimized and parallelized by means of the Pluto

automatic parallelization and optimization tool. Algorithm 2 illustrates the code transformation, using the unimodular function, of all statements in which a triangular matrix is referred to. For simplicity, we suppose without loss of generality that the statement contains only one reference (x, y) to a triangular matrix. This reference is transformed in 2d-packed format as follows:

$$f_t(x,y) = \begin{cases} (-x+N-1, -y+N) & \text{if } x > \frac{N}{2} \wedge y > \frac{N}{2} \\ (x,y) & \text{else} \end{cases} \quad (1)$$

This piecewise unimodular transformation function can be rewritten in the matrix representation as:

$$f_t(x,y) = \begin{cases} \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} N-1 \\ N \end{pmatrix} & \text{if } x > \frac{N}{2} \wedge y > \frac{N}{2} \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \text{else} \end{cases}$$

ALGORITHM 1: Storage in 2d-packed format for triangular matrices.

Input: Triangular matrices Mat_i of size N

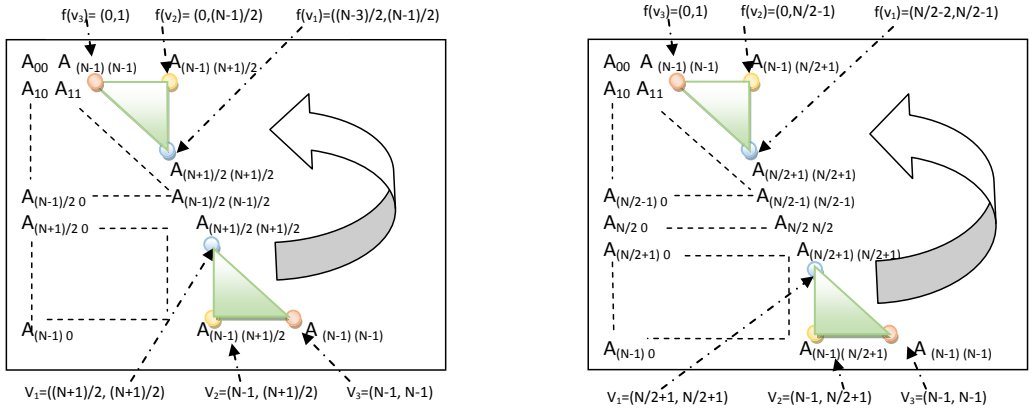
```
for RowInd ← 0 to N − 1 do
    for ColInd ← 0 to RowInd − 1 do
        if RowInd > N/2 and ColInd > N/2 then
            StorMat_i(N − RowInd − 1, N − ColInd) ← Mat(RowInd, ColInd);
        else
            StorMat_i(RowInd, ColInd) ← Mat(RowInd, ColInd);
        end
    end
end
```

ALGORITHM 2: Triangular-matrix code transformation for the 2d-packed storage format.

Input: Triangular matrices Mat_i of order N

```
for any statement containing references Mat_i(x_j, y_j) to triangular matrices do
    replace any reference Mat_i(x_j, y_j) with StorMat_i(z_j, w_j) where:
    if x_j > N/2 and y_j > N/2 then
        (z_j, w_j) = (N − x_j − 1, N − y_j);
    else
        (z_j, w_j) = (x_j, y_j);
    end
end
```

Proof. In order to prove that our transformation function for triangular matrices f_t is valid, we need to demonstrate that it is a bijection: any element belonging to the lower right triangle in Figure 8 has one and only one image belonging to the upper triangle. The other non zero elements of the matrix stay untouched. Since the transformation of the first elements is unimodular, it suffices to prove that the images of the vertices of the lower triangle are exactly the vertices of the upper triangle (see Figure 8), where the coordinates of the vertices are the indices of the corresponding matrix elements.



(a) Transformation for odd triangular matrices.

(b) Transformation for even triangular matrices.

Fig. 8. Matrix transformation for triangular matrices.

When N is odd, the vertices of the lower triangle are: $V_1 = (\frac{N+1}{2}, \frac{N+1}{2})$, $V_2 = (N-1, \frac{N+1}{2})$ and $V_3 = (N-1, N-1)$. The transformation of the vertices V_1, V_2 and V_3 using the unimodular function (1) are given by $f_t(V_1), f_t(V_2)$ and $f_t(V_3)$, respectively:

$$\begin{array}{lcll} f_t(V_1) & = & (-\frac{N+1}{2} + N - 1, -\frac{N+1}{2} + N) & = & (\frac{N-3}{2}, \frac{N-1}{2}) \\ f_t(V_2) & = & (-(N-1) + N - 1, -\frac{N+1}{2} + N) & = & (0, \frac{N-1}{2}) \\ f_t(V_3) & = & (-(N-1) + N - 1, -(N-1) + N) & = & (0, 1) \end{array}$$

One can notice that these are exactly the vertices of the upper triangle (Figure 8a), which is disjoint from the other non zero elements, so the odd case is proven.

When N is even, the vertices of the lower triangle are: $V_1 = (\frac{N}{2} + 1, \frac{N}{2} + 1)$, $V_2 = (N-1, \frac{N}{2} + 1)$ and $V_3 = (N-1, N-1)$. The transformation of these vertices using the same function (1) are given by $f_t(V_1), f_t(V_2)$ and $f_t(V_3)$, respectively:

$$\begin{array}{lcll} f_t(V_1) & = & (-(\frac{N}{2} + 1) + N - 1, -(\frac{N}{2} + 1) + N) & = & (\frac{N}{2} - 2, \frac{N}{2} - 1) \\ f_t(V_2) & = & (-(N-1) + N - 1, -(\frac{N}{2} + 1) + N) & = & (0, \frac{N}{2} - 1) \\ f_t(V_3) & = & (-(N-1) + N - 1, -(N-1) + N) & = & (0, 1) \end{array}$$

Again, we can see that they are the vertices of the upper triangle (Figure 8b), which is disjoint from the other untouched non-zero elements. □

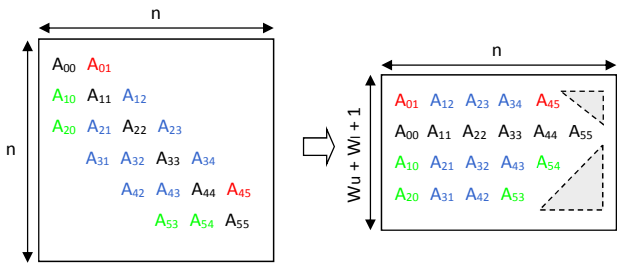


Fig. 9. Banded matrix storage in 2d-packed format.

4.2 2d-packed optimization technique for banded matrices

We propose a similar optimization approach to be applied to banded matrices. In this case, we store the $N^2 - ((N - W_u)^2 + (N - W_l)^2 - 2N - W_u - W_l)/2$ non-zero elements of a square banded matrix of order N in a 2d-packed data structure of $W_l + W_u + 1$ rows, and rows of different sizes (see Figure 9). This structure is slightly different from the one implemented in the LAPACK library [4]. Indeed, all the upper-band elements are shifted to the left in order not to store any zero element (using dynamic allocation). Figure 10 shows a C code dynamically allocating memory for this new 2d-packed data structure. Algorithms 3 and 4 illustrate, respectively, the way the non-zeros elements of banded matrices are stored in 2d-packed format, and the code transformation for all statements in which the banded matrices are referred to.

ALGORITHM 3: Storage in 2d-packed format for banded matrices.

Input: Banded matrices Mat_i of order N and Upper bandwidth W_u and Lower bandwidth W_l

```
1  for RowInd ← 0 to N − 1 do
2      for ColInd ← 0 to N − 1 do
3          if ColInd <= (RowInd + W_u) and ColInd >= (RowInd − W_l) then
4              if RowInd > ColInd then
5                  StorMat_i(W_u + RowInd − ColInd, ColInd) ← Mat(RowInd, ColInd);
6              else
7                  StorMat_i(W_u + RowInd − ColInd, RowInd) ← Mat(RowInd, ColInd);
8              end
9          end
10     end
11 end
```

ALGORITHM 4: Banded-matrix code transformation for the 2d-packed storage format.

Input: Banded matrices Mat_i of order N and Upper bandwidth W_u and Lower bandwidth W_l

```
1  for any statement containing references Mat_i(x_j, y_j) to banded matrices do
2      replace any reference Mat_i(x_j, y_j) with StorMat_i(z_j, w_j) Where:
3      if x_j > y_j then
4          (z_j, w_j) = (W_u + x_j − y_j, y_j);
5      else
6          (z_j, w_j) = (W_u + x_j − y_j, x_j)
7      end
8  end
```

Once again, we consider (without loss of generality) that we are given a statement containing only one reference (x, y) to a banded matrix. This reference is transformed as follows:

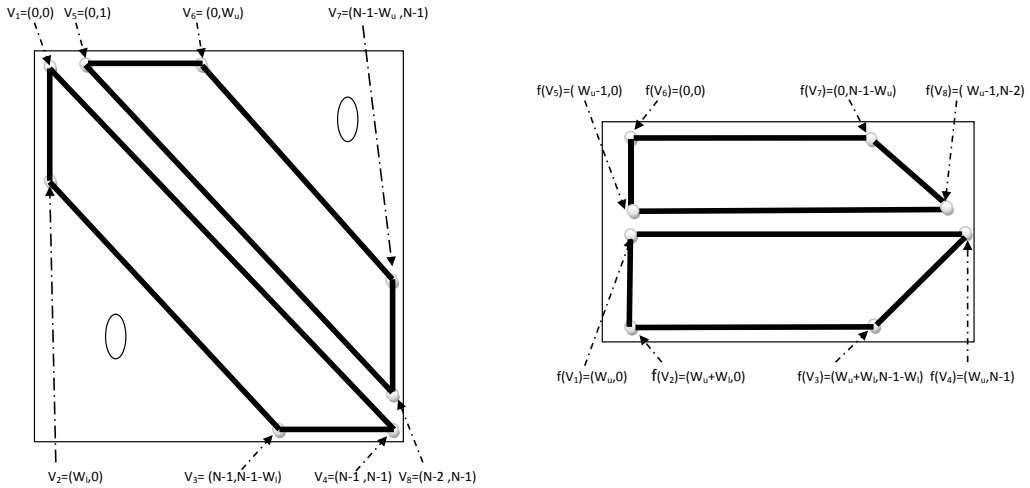
$$f_b(x,y) = \begin{cases} (W_u + x - y, y) & \text{if } x > y \\ (W_u + x - y, x) & \text{else} \end{cases} \tag{2}$$

The matrix representation of this piecewise unimodular transformation is:

$$f_b(x,y) = \begin{cases} \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} W_u \\ 0 \end{pmatrix} & \text{if } x > y \\ \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} W_u \\ 0 \end{pmatrix} & \text{else} \end{cases}$$

```
1    double **allocBandMat(int n, int Wu, int Wl){
2    double **mat; int i;
3     mat= malloc((Wu+Wl+1)*sizeof(int *));
4     for (i=0; i<=Wu; i++)
5        mat[Wu-i]=calloc((n-i),sizeof(double));
6     for(i=1; i<= Wl; i++)
7        mat[Wu+i]=calloc((n-i),sizeof(double));
8     return mat;
9   }
```

Fig. 10. Dynamic allocation for banded-matrices in 2d-packed format.



(a) Original banded matrix.

(b) Banded matrix in 2d-packed format.

Fig. 11. Matrix transformation for banded matrices.

Proof. Similarly to triangular matrices case, we can prove the validity of our transformation for banded matrices f_b by demonstrating that any element belonging to the upper (respectively lower) trapezoid of Figure 11a has one and only one image belonging to the upper (respectively lower) trapezoid of Figure 11b. Because of the unimodularity of our two transformations, it suffices to prove that the images of the two trapezoids are disjoint, or equivalently, that the vertices of the upper and lower trapezoids of Figure 11a correspond exactly to the vertices of the upper and lower disjoint trapezoids of Figure 11b, which is done in the same way as in the previous proof. The coordinates of the vertices are given in Figure 11. □

5 EXPERIMENTAL RESULTS

To evaluate our approach, we have combined the new 2d-packed format transformation with the Pluto source-to-source compiler. Pluto is used in order to automatically parallelize and vectorize programs, in addition to memory accesses optimization through data locality improvement. We have applied our approach to optimize and parallelize six double-precision linear algebra computation kernels with the different matrix types shown in Table 1. Notice that some matrix computations, such as Cholesky and sspfa, cannot be applied to non-triangular matrices, and thus do not appear in

Table 1. Matrix computation kernels

Computation	Matrix Type	Designation
Cholesky Factorization	Banded Triangular	`choleskyBT`
	Triangular	`choleskyTri`
Matrix Matrix Multiplication	Banded Triangular	`matmulBT`
	Banded	`matmulBand`
	Triangular	`matmulTri`
Matrix Vector Multiplication	Banded Triangular	`mvBT`
	Banded	`mvBand`
	Triangular	`mvTri`
Matrix Matrix Solver	Banded Triangular	`solvematBT`
	Triangular	`solvematTri`
Matrix Vector Solver	Banded Triangular	`solvevectBT`
	Triangular	`solvevectTri`
sspfa Factorisation	Banded Triangular	`sspfaBT`
	Triangular	`sspfaTri`

Table 2. Code versions

Code version	Description
Original	Original sequential code handling full square matrices
Original + Pluto	The original code automatically optimized and parallelized by Pluto
LPF	The sequential code handling sparse matrices in Linear Packed Format
2d-packed	The sequential code handling sparse matrices in 2d-packed format
2d-packed + Pluto	The 2d-packed code automatically optimized and parallelized by Pluto
Parallel MKL	The icc parallellized and optimized MKL routine handling full square matrices

this table. Those kernels are the same than the ones evaluated by Cui et al. [18], plus the Cholesky factorization that was evaluated by Gustavson et al. [24]. To show the effectiveness of our approach, we have compared its performance against a set of existing methods and libraries dealing with linear algebra computations. The comparisons have been performed between different, but semantically equivalent, code versions as shown in Table 2. All programs were first tested for checking the output matrices correctness, before being run for execution time measurements (without printing out large amounts of data).

For all our experiments, we have used a dual socket 2x10 cores (and 40 threads) Intel® Xeon® CPU E5-2650 v3 @ 2.30 GHz running Linux 4.4.0. We have compiled our programs using icc 17.0.0 and gcc 5.4.0, with options `-O3 -march=native`. The programs calling MKL routines are compiled using icc with the additional flags `-mkl -parallel`. In addition, Pluto version 0.11.4, with options `--parallel --tile` has been called to automatically parallelize, tile and optimize the original and the 2d-packed codes. The LPF (Linear Packed Format) codes are sequential, since they cannot be automatically parallelized by Pluto, because in LPF storage the access functions are non-linear. Each program has been run five times and the execution times of the computation kernels measured

Table 3. MKL routine calls

Benchmark	name of MKL routine	Percentage of data processed by the 2d-packed version
choleskyBT	LAPACKE_dpotrf	44%
choleskyTri	LAPACKE_dpotrf	100%
matmulBT	cblas_dgemm	22%
matmulBand	cblas_dgemm	19%
matmulTri	cblas_dgemm	50%
mvBT	cblas_dgemv	22%
mvBand	cblas_dgemv	19%
mvTri	cblas_dgemv	50%
solvematBT	cblas_dtrsm	44%
solvematTri	cblas_dtrsm	100%
solvevectBT	cblas_dtrsv	44%
solvevectTri	cblas_dtrsv	100%

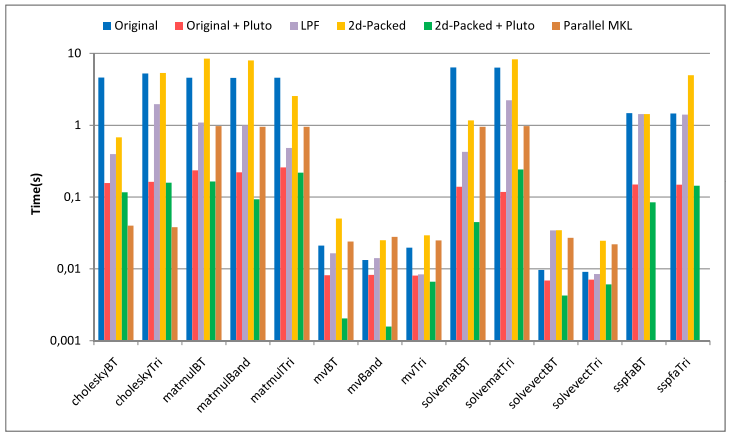
using `gettimeofday()`. The two extremal measurements have been eliminated and the average of the three remaining ones is reported.

Figures 12 and 13 show the execution times obtained using the icc and gcc compilers, respectively. The icc runtime figures plot six bars corresponding to the six code versions shown in Table 2, except for sspfa which has no equivalent routine in the MKL Library. The gcc runtime figures plot only five bars: there is no MKL library provided with gcc. All the Y-axes are in logarithmic scale because of the important variation in their values. In both figures, subfigures (a), (b) and (c) illustrate the runtime of the different benchmarks for small (N=2000 or 4000), medium (N=4000 or 8000) and large (N=8000 or 16000) matrices, respectively. The first value of N is used for all $O(N^3)$ algorithms, the second one for the two $O(N^2)$ algorithms: matrix-vector product and matrix-vector solver.

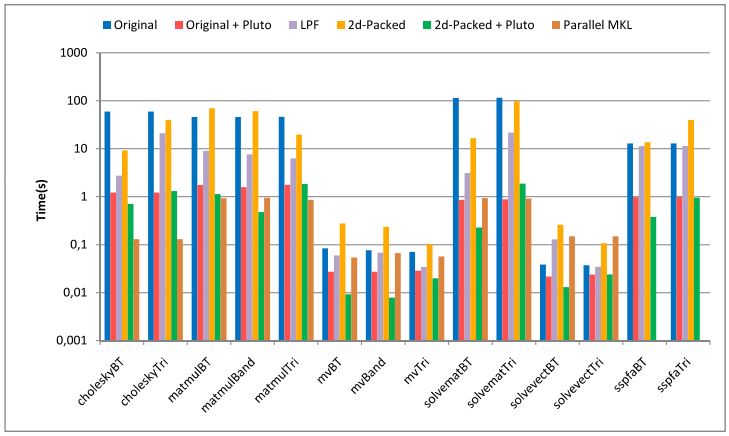
In all figures, one can notice that the highest times are those corresponding to the original and 2d-packed codes. This is because they were not parallelized and tiled, and because of the storage of a large amount of unneeded zeros in the first case, and to the tests and extra-arithmetic operations in the array subscripts in the second case. When these two codes are optimized and parallelized by Pluto, their performances are significantly enhanced. They are also better compared to the LPF code, which cannot be automatically parallelized. Notice that, in addition to their low memory storage benefit, the performances of our 2d-packed optimized codes (2d-packed + pluto) are better or equivalent to those of the original optimized codes (original + pluto) for almost all of them. The reason behind this delta is that, first, the 2d-packed codes access globally less memory, and as a consequence there is less pressure on the memory hierarchy and on the caches in particular; second, some of the original codes perform computations on unneeded zeros, that have been removed in the 2d-packed codes. On average over all those experiments, the 2d-packed parallelized codes perform better than the original parallelized codes with a ratio of:

- 1.64x for gcc,
- 1.78x for icc.

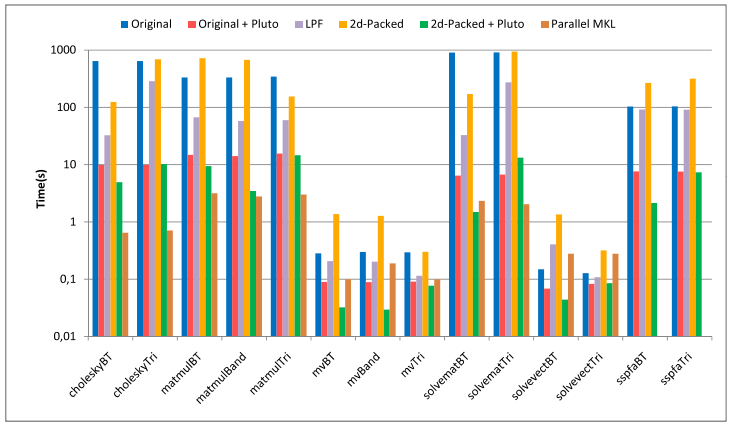
The comparison of the 2d-packed optimized codes against icc parallelized non-packed MKL routines (Figure 12) shows that our method outruns the handwritten routines from MKL in ten out of twelve cases for small matrices, and for medium and large matrices the performances are similar, with a high variance. The MKL routines are consuming more memory than our parallelized 2d-packed codes, since they handle matrices in their full square format. On the other hand, since



(a) Small matrices (N=2000; 4000 for $O(N^2)$ benchmarks).

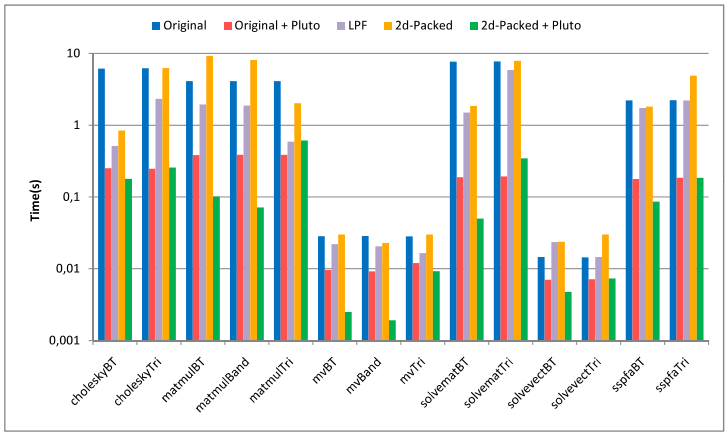


(b) Medium matrices (N=4000; 8000 for $O(N^2)$ benchmarks).

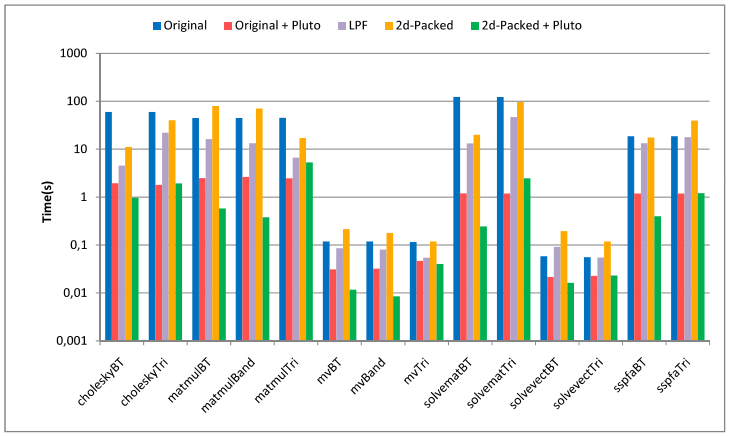


(c) Large matrices (N=8000; 16000 for $O(N^2)$ benchmarks).

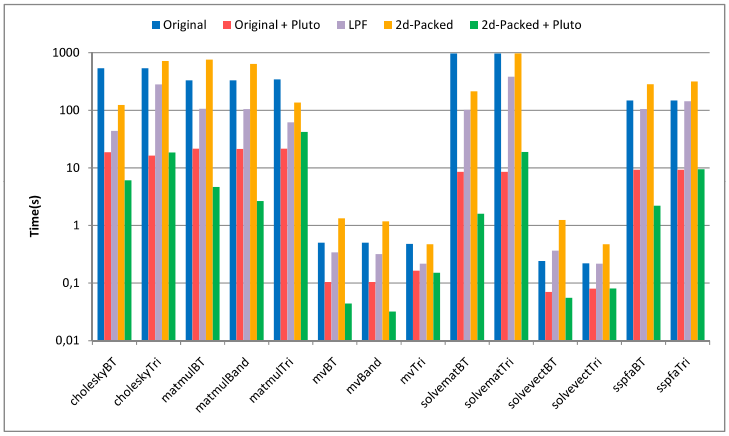
Fig. 12. Execution time using the ICC Compiler.



(a) Small matrices (N=2000; 4000 for $O(N^2)$ benchmarks).



(b) Medium matrices (N=4000; 8000 for $O(N^2)$ benchmarks).



(c) Large matrices (N=8000; 16000 for $O(N^2)$ benchmarks).

Fig. 13. Execution time using the GCC compiler.

they perform computations on full or triangular matrices, we have to report the amount of extra data that is not processed by the 2d-packed version, but is processed by the MKL routines. This is summarized in Table 3, along with the routine name that was used for each benchmark. Notice that we report 100% of data processed by the 2d-packed triangular version in some cases, when the input of the MKL routine is a triangular or a symmetric matrix; 50% in the other cases, when the input is a full square matrix. The outcome of this experiment is that the 2d-packed codes often compete against the non sparse optimized by hand MKL code.

We also ran the three kernels proposed by Gustavson et al. [24] for triangular matrices (cholesky fatorization: `LAPACKE_dpftrf`, inverse: `LAPACKE_dtftri`, and solve cholesky: `LAPACKE_dpftrs`), and observed that they perform about the same as the ones using the full matrix format in MKL (`LAPACKE_dpotrf`, `LAPACKE_dpotri`, `LAPACKE_dpotrs`). Since we compare to the latter directly for the cholesky factorization, we did not present a further comparison to Gustavson et al.'s three kernels.

Not reported in those figures, we have also tried to parallelize the LPF codes using the icc auto-parallelizer (with options -parallel -O3 -march=native), and it comes out that icc is not able to parallelize those codes, most probably because of the non-linear accesses to the linear packed arrays and the resulting complex dependence analysis. The execution times using the -parallel option are the same than the reported sequential execution times, except for two very short matrix-vector product codes (mvBT and mvBand).

According to the above set of experiments, we conclude that our 2d-packed format transformation combined with Pluto is able to significantly enhance the performances of those sparse matrix computations. This is not the case for the linear packed format, whose automatic optimization and parallelization is quite difficult.

6 CONCLUSION

In this paper, we have introduced new data structure storage schemes for triangular and banded matrices, which allow to optimize matrix computations and achieve better runtime performance on multicore architectures. The access functions to the new 2d-packed format structures are piecewise affine functions. As a consequence, the transformed code can be optimized and parallelized through automatic polyhedral tools, such as the Pluto source-to-source compiler. Our experimental results show a great performance enhancement for triangular and banded matrix computation benchmarks, compared to the LPF storage scheme typically used for sparse matrices storage.

Our approach is more general than the state-of-the-art since it can be applied to any matrix computation. It could be implemented as an automatic tool that transforms programs handling banded and triangular matrices into programs handling 2d-packed format structures, which allows to take full advantage of the well-established polyhedral compilers. Our goal in this paper was to show that a standard *out-of-the-box* polyhedral compiler enhances the performance of banded and triangular 2d-packed matrix computations. But a future improvement could be to fine-tune Pluto in order to further improve the performance of the generated codes, and in particular to achieve better SIMDization since those codes are very sensitive to tiling and to vectorization.

REFERENCES

[1] Ramesh C. Agarwal, Fred G. Gustavson, Mahesh V. Joshi, and Mohammad Zubair. 1995. A Scalable Parallel Block Algorithm for Band Cholesky Factorization. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1995, San Francisco, California, USA, February 15-17, 1995*. 430–435.

[2] Äke Björck. 2015. *Numerical Methods in Matrix Computations*. Springer International Publishing.

[3] Bjarne Stig Andersen, Jerzy Waśniewski, and Fred G. Gustavson. 2001. A Recursive Formulation of Cholesky Factorization of a Matrix in Packed Storage. *ACM Trans. Math. Softw.* 27, 2 (June 2001), 214–244. https://doi.org/10.1145/383738.383741

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[5] Howard Anton and Chris Rorres. 2014. *Elementary Linear Algebra: Applications Version* (eleventh ed.). Wiley.

[6] Athanasios Athanasios Konstantinidis and Paul H. J. Kelly. 2011. More Definite Results from the PluTo Scheduling Algorithm. In *1st International Workshop on Polyhedral Compilation Techniques (IMPACT)*, C. Alias and C. Bastoul (Eds.). Chamonix, France. http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-02.pdf

[7] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2009. Compiler-assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, New York, NY, USA, 219–228. https://doi.org/10.1145/1504176.1504209

[8] Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, Washington, DC, USA, 7–16. https://doi.org/10.1109/PACT.2004.11

[9] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

[10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model is More Widely Applicable Than You Think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer-Verlag, Berlin, Heidelberg, 283–303. https://doi.org/10.1007/978-3-642-11970-5_16

[11] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 2014. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, New York, NY, USA, 253–260. https://doi.org/10.1145/2591635.2667174

[12] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43, 6 (June 2008), 101–113. https://doi.org/10.1145/1379022.1375595

[13] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2007. *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer*. Technical Report OSU-CISRC-10/07-TR70. The Ohio State University.

[14] Uday Kumar Reddy Bondhugula. 2008. *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model*. Ph.D. Dissertation. Ohio State University, Columbus, OH, USA. Advisor(s) Sadayappan, P. AAI3325799.

[15] Aydin Buluc and John R. Gilbert. 2008. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP '08)*. IEEE Computer Society, Washington, DC, USA, 503–510. https://doi.org/10.1109/ICPP.2008.45

[16] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Comput.* 35, 1 (Jan. 2009), 38–53. https://doi.org/10.1016/j.parco.2008.10.002

[17] Huimin Cui, Jingling Xue, Lei Wang, Yang Yang, Xiaobing Feng, and Dongrui Fan. 2012. Extendable Pattern-oriented Optimization Directives. *ACM Trans. Archit. Code Optim.* 9, 3, Article 14 (Oct. 2012), 37 pages. https://doi.org/10.1145/2355585.2355587

[18] Huimin Cui, Qing Yi, Jingling Xue, and Xiaobing Feng. 2013. Layout-oblivious Compiler Optimization for Matrix Computations. *ACM Trans. Archit. Code Optim.* 9, 4, Article 35 (Jan. 2013), 20 pages. https://doi.org/10.1145/2400682.2400694

[19] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 1–17. https://doi.org/10.1145/77626.79170

[20] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G.W. Stewart. 1979. *LINPACK Users' Guide*. pub-SIAM. 320 pages.

[21] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, Part 1 : one dimensional time. *Int. J. of Parallel Programming* 21, 5 (October 1992), 313–348.

[22] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, Part 2 : multidimensional time. *Int. J. of Parallel Programming* 21, 6 (December 1992).

[23] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Int. J. Parallel Program.* 34, 3 (June 2006), 261–317. https://doi.org/10.1007/s10766-006-0012-3

[24] Fred G. Gustavson, Jerzy Waśniewski, Jack J. Dongarra, and Julien Langou. 2010. Rectangular Full Packed Format for Cholesky's Algorithm: Factorization, Solution, and Inversion. *ACM Trans. Math. Softw.* 37, 2, Article 18 (April 2010), 21 pages. https://doi.org/10.1145/1731022.1731028

[25] Ziang Hu, Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. 2006. *Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences*. Springer Berlin Heidelberg, Berlin, Heidelberg, 134–144. https://doi.org/10.1007/11823285_14

[26] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. 1967. The Organization of Computations for Uniform Recurrence Equations. *J. ACM* 14, 3 (July 1967), 563–590. https://doi.org/10.1145/321406.321418

[27] H. T. Kung and Jaspal Subhlok. 1991. A New Approach for Automatic Parallelization of Blocked Linear Algebra Computations. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 122–129. https://doi.org/10.1145/125826.125898

[28] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. 2002. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* 6, 1 (01 Feb. 2002), 4–15.

[29] G. M. Megson and X. Chen. 1997. *Automatic parallelization for a class of regular computations*. World Scientific.

[30] A. P. Mullhaupt and K. S. Riedel. 2001. Banded matrix fraction representation of triangular input normal pairs. *IEEE Trans. Automat. Control* 46, 12 (Dec 2001), 2018–2022. https://doi.org/10.1109/9.975512

[31] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2016. Adaptive Multi-level Blocking Optimization for Sparse Matrix Vector Multiplication on GPU. *Procedia Computer Science* 80 (2016), 131 – 142. https://doi.org/10.1016/j.procs.2016.05.304

[32] Jeff Parkhurst, John Darringer, and Bill Grundmann. 2006. From Single Core to Multi-core: Preparing for a New Exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design (ICCAD '06)*. ACM, New York, NY, USA, 67–72. https://doi.org/10.1145/1233501.1233516

[33] Diogo N. Sampaio, Louis-Noël Pouchet, and Fabrice Rastello. 2017. Simplification and Runtime Resolution of Data Dependence Constraints for Loop Transformations. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3079079.3079098

[34] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-Model Guided Loop-Nest Auto-Vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. IEEE Computer Society, Washington, DC, USA, 327–337. https://doi.org/10.1109/PACT.2009.18

[35] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 31, 11 pages. http://dl.acm.org/citation.cfm?id=1413370.1413402

[36] Qing Yi. 2011. Automated Programmable Control and Parameterization of Compiler Optimizations. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 97–106. http://dl.acm.org/citation.cfm?id=2190025.2190057

[37] Ling Zhuo and Viktor K. Prasanna. 2005. Sparse Matrix-Vector Multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays (FPGA '05)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1046192.1046202

Received May 2017; revised November 2017; accepted November 2017