International Conference on Automation, Control, Engineering and Computer Science (ACECS'14)
Proceedings - Copyright IPCO-2014, pp.87-94
ISSN 2356-5608

Dense-Sparse Matrix Multiplication : Algorithms and Performance Evaluation

S. Ezouaoui , O. Hamdi-Larbi, Z. Mahjoub
University of Tunis El Manar - Faculty of Sciences of Tunis
University Campus 2092 El Manar
Tunis, Tunisia

Abstract—. **In this paper, we address the dense-sparse matrix product (DSMP) problem i.e. where the first matrix is dense and the second is sparse. We first present initial versions of loop nest structured algorithms corresponding to the most used sparse matrix storing formats i.e. DNS, CSR, CSC and COO. Afterwards, we derive several versions obtained by applying loop interchange techniques, loop invariant motion and loop unrolling on the previous loop nest algorithms. Theoretical multifold comparisons are then made between the different designed versions. Our contribution is validated through a series of experiments achieved on a set of sparse matrices with different sizes and densities.**

Keywords— Algorithm complexity; compressed/storage format; loop nest optimization; performance evaluation; sparse matrix product

I. INTRODUCTION

Several scientific applications often use kernels performing computations on large size sparse matrices e.g. in semi-conductors, robotics, image processing, networks and graphs, molecular dynamics etc [1], [2], [3], [4], [5], [6]. Most problems in these fields reduce to sparse linear algebra kernels [2].

Many works have been devoted to the sparse computing problem such as Sparse Matrix Product (SMP) [1], [4], [6], Sparse Matrix Vector Product (SMVP) [2], [5], [7] and Sparse-Dense Matrix Product (SDMP) [8], [9], [10], [11]. The symmetric case of the SDMP problem i.e. Dense-Sparse matrix product (DSMP) where the first is dense and the second is sparse is also an important kernel, especially for computing a sparse matrix chain product [12], [13] as well for building Peano space-filling curves [8]. However, DSMP has not been enough studied in literature. To our knowledge, the impact of sparse storing formats, matrix access/storing modes and data locality have not been studied so far.

This paper constitutes a symmetric investigation related a previous work [9] where we studied SDMP optimization. Let us first recall that processing large sparse matrices requires, for reasons of space-time complexity reduction, the use of compressing (or storing) formats (SCF). These latter may be either general i.e. fitting any sparse structure e.g. DNS (DeNSe) where both zero and nonzero elements are stored, CSR (Compressed Sparse Row), CSC (Compressed Storage Column) and COO (COOrdinate)…), or particular such as MSR (Modified Storage Row), BND (BaNDed), DIA (Diagonal)… , [5] , [6], [7].

Our aim here is to determine the best SCF for the DSMP i.e. leading to the best performances. For this purpose, we derived a series of algorithms corresponding to four SCF's, namely DNS, CSR, CSC and COO.

The remainder of the paper is organized as follows. In section II, a very brief survey on SCF's is given. Four SCF's being chosen, Section III is devoted to a detailed description of two successive sets of algorithms for DSMP where the first involves initial loop nest structured algorithms (LNSA), from which we derive a second set of LNSA's by applying specific loop nest transformation techniques. Section IV is devoted to an experimental study in order to validate our theoretical contribution. Finally, we conclude our study and present some perspectives in section V.

II. SPARSE MATRICES AND COMPRESSION FORMATS

Let us recall that a matrix is called sparse if it has a large (resp. weak) number of zero (resp. nonzero) elements [5], [7]. Let NNZ be the number of nonzero elements. As previously mentioned, processing sparse matrices requires using particular SCF's restricted to the nonzero elements.

A sparse matrix can have various structures according to the locations of its nonzero elements. The structure of a sparse matrix may be either regular e.g. triangular, diagonal, constant band, etc; or irregular (called also general) e.g. variable band, random, etc [5], [7], [14]. In this paper, we are interested in four most used storage formats namely DNS, CSR, CSC and COO.

We recall that storing a sparse matrix, say B of size N with NNZ nonzero elements, the CSR data-structure consists of three arrays B, JB and IB i.e. a real array B(1:NNZ) to store row-wise the nonzero elements of B, an integer array JB(1:NNZ) to store the column positions of the elements in the real array B, and finally, a pointer array IB(1:n+1), the i-th entry of which points to the beginning of the i-th row in arrays B and JB [5] , [7].

The CSC is similar to CSR except that the nonzero elements are stored column-wise in the first array, a row index is stored for each element, and column pointers are stored. So, CSC is specified by three arrays, denoted B, IB and JB, where IB stores the row indices of each nonzero, JB stores the index of the elements in B which start a column of the matrix [7].

As to the COO format, it also consists of three arrays, each of which is of size NNZ : an array B of floats containing the nonzero elements, an array IB (resp. JB) of integers containing their row (resp. column) indices [5].

III. Theoretical Study

It has to be firstly recalled, that in the SDMP case, denoted C=AB where A is sparse and B is dense, the algorithms we designed could be easily derived from known algorithms for Sparse Matrix Dense Vector Product (SMDVP) algorithms [9]. As to the DSMP case (where A is dense and B is sparse) in which we are interested, a direct approach will be adopted since algorithms for the Dense Matrix Sparse Vector Product (DMSVP) are of no use. Let us add that in both SDMP and DSMP cases, 2N*NNZ flops are required, NNZ being the number of nonzero elements of the sparse matrix and N its size [9].

A. DNS Format

Given the standard algorithm, structured in a perfect 3-loop nest denoted IJK, for computing the product of two square dense matrices, we include some modifications consisting in logical tests and scalar replacements [9]. The aim is to avoid useless operations and reduce the number of accesses to matrix B (see algorithm (a) below). Clearly, five other versions may be derived by applying the loop interchange (LI) technique i.e. IKJ, KJI, KIJ, JKI, JIK. This transformation has an impact on data locality and may modify the loop nest body kernel (see Table I, where R is for Row and C for Column) [3], [6], [9].

Considering the initial version IJK and since the logical test is done on element B(k,j), we'll keep both JKI (see algorithm (b) below) and KJI versions and apply loop invariant motion technique (LIM) [15] in order to reduce the number of logical tests (see algorithm (c) below). Remark that the loop nest body kernel is GAXPY-C in version JKI (resp. AXPY-C in version KJI).

```
DNS_IJK
  DO i=1, N
     DO j=1, N
        DO k=1, N
           s=B(k,j)
           IF (s≠0) THEN
                       C(i,j)=C(i,j) + A(i,k)*s
           ENDIF
        ENDDO
     ENDDO
  ENDDO
```

(a)

```
DNS_JKI
  DO j=1, N              / first level /
     DO k=1, N           / second level /
        DO i=1, N        / third level /
           s=B(k,j)
           IF (s≠0) THEN
                       C(i,j)=C(i,j) + A(i,k)*s
           ENDIF
        ENDDO
     ENDDO
  ENDDO
```

(b)

```
DNS_JKI_V1
  DO j=1, N
     DO k=1, N           / second level /
        s=B(k,j)
        IF (s≠0) THEN
                    DO i=1, N
                       C(i,j)=C(i,j) + A(i,k)*s
                    ENDDO
        ENDIF
     ENDDO
  ENDDO
```

(c)

Table I recapitulates a comparative study on the whole 6 versions including number of accesses, number of tests, access mode and body kernel.

We can notice, from Table I, that both versions KJI_V1 and JKI_V1 lead to less accesses and logical tests i.e. N^2 (resp. NNZ) accesses to B (resp. A) instead of N^3 (resp. N*NNZ) for the four other versions.

Let us add that the JKI kernel is GAXPY-C i.e. same column-wise access for the three matrices whereas the KJI kernel is AXPY-C i.e. A and C are accessed column-wise and B row-wise. So, in order to reduce cache misses thus improve performances, the three matrices have to be processed according to their storing mode (i.e. either row-wise such in a C environment or column-wise such in a Fortran environment). This storing-access mode fitting leads to better data locality [16].

Consequently, versions KJI (A and C : column-wise access, AXPY-C kernel) and JKI (A, B and C : column-wise access, GAXPY-C kernel) are better in a Fortran environment. It must be emphasized that in a C environment, the best version from a data locality point of view is IKJ. Unfortunately, for this latter, the LIM technique cannot be applied. To resume, since we worked in a C environment (see section IV), we have two versions (JKI and KJI) where storing and access modes conflict but the number of logical tests (nlt) is minimized, and a third version (IKJ) where the two modes fit but the nlt is maximized.

International Conference on Automation, Control, Engineering and Computer Science (ACECS'14)
Proceedings - Copyright IPCO-2014, pp.87-94
ISSN 2356-5608

TABLE I. DNS RECAPITULATION TABLE

Version	**A**			**B**					**C**	**Kernel**
	Initial version	V_1	*Access mode*	*Initial version*		V_1		*Access mode*	*Access mode*	
	#Access	*#Access*		*#Access*	*#Test*	*#Access*	*#Test*			
IJK	N*NNZ	N*NNZ	R	N^3	N^3	N^3	N^3	C	R	DOT-R
JIK	N*NNZ	N*NNZ	R	N^3	N^3	N^3	N^3	C	C	DOT-C
KIJ	N*NNZ	N*NNZ	C	N^3	N^3	N^3	N^3	R	R	AXPY-R
KJI	N*NNZ	**NNZ**	C	N^3	N^3	$\mathbf{N^2}$	$\mathbf{N^2}$	R	C	AXPY-C
IKJ	N*NNZ	N*NNZ	R	N^3	N^3	N^3	N^3	R	R	GAXPY-R
JKI	N*NNZ	**NNZ**	C	N^3	N^3	$\mathbf{N^2}$	$\mathbf{N^2}$	C	C	GAXPY-C

TABLE II. SCF'S COMPARISON

Format	**Version**	**A**	**B**							**C**	**Kernel**
		MA[a]	*Initial version (#Access)*			V_1 *(#Access)*			*MA*[a]	*MA*[a]	
			IB	*JB*	*B*	*IB*	*JB*	**B**			
CSR	**JKI**	C	2N*NNZ	N*NNZ	N*NNZ	**2N**	**NNZ**	N*NNZ	R	C	AXPY-C
	JIK	C	2N*NNZ	N*NNZ	N*NNZ	**2N**	N*NNZ	N*NNZ	R	R	AXPY-R
	IJK	R	2N*NNZ	N*NNZ	N*NNZ	$\mathbf{2N^2}$	N*NNZ	N*NNZ	R	R	GAXPY-R
CSC	**IJK**	R	N*NNZ	2N*NNZ	N*NNZ	N*NNZ	$\mathbf{2N^2}$	N*NNZ	C	R	DOT-R
	JIK	R	N*NNZ	2N*NNZ	N*NNZ	N*NNZ	**2N**	N*NNZ	C	C	DOT-C
	JKI	C	N*NNZ	2N*NNZ	N*NNZ	**NNZ**	**2N**	**NNZ**	C	C	GAXPY-C
COO	**IK**	R	N*NNZ	2N*NNZ	N*NNZ	N*NNZ	N*NNZ	N*NNZ	R	R	GAXPY-R
	KI	C	N*NNZ	2N*NNZ	N*NNZ	**NNZ**	**NNZ**	**NNZ**	R	C	AXPY-C

a. AM : Access mode (Row/Column)

B. CSR, CSC and COO versions

From the initial versions, denoted CSR_JKI, CSC_IJK and COO_IK, we derived other versions by apply improving techniques such as scalar replacement and loop interchange in order to ensure better data locality (see (e), (f) and (g) .

```
CSR_JKI
 DO j=1,N
      ibj= IB(j); ibj1= IB(j+1)-1
      DO k=ibj,ibj1
           DO i=1,N
                C(i,JB(k))= C(i,JB(k))+A(i,j)*B(k)
           ENDDO
      ENDDO
 ENDDO                    (e)

CSC_IJK
  DO i=1,N
      DO j=1,N
           jbj= JB(j), jbj1= JB(j+1)-1
           DO k=jbj, jbj1
                C(i,j)=C(i,j)+A(i,IB(k))*B(k)
           ENDDO
      ENDDO
  ENDDO               (f)

COO_KI
 DO k=1,NNZ
      jbk= JB(k), ibk=IB(k)
      DO i=1,N
           C(i,jbk)= C(i,jbk)+ A(i,ibk)*B(k)
      ENDDO
 ENDDO                    (g)
```

Given these three versions, the LI technique permitted to derive five other versions i.e. CSR_JIK, CSR_IJK, CSC_JKI, CSC_JIK and COO_KI (see Table II).

Hence, from three initial versions (see Table II), we derived 18 other i.e. 7 for CSR, 7 for CSC and 3 for COO (see Table II). Notice that for any version, the number of accesses to matrix A is equal to N*NNZ. It is easy to deduce from comparative intra-SCF versions that CSR_IJK_V1, CSC_IJK_V1 and COO_IK_V1 would (theoretically) be the best in a C environment (row-wise storing). This is due to two reasons since they perform (i) 2-3 (out of 3) row accesses to A, B, and C and (ii) less accesses to these arrays. We precise that in the COO format, we have chosen a row-wise storing for B.

IV. EXPERIMENTAL STUDY

In order to evaluate the practical performances of the derived versions and validate our theoretical study, we achieved a series of experimentations on a set of randomly generated matrices of different sizes and densities. Indeed, we have chosen 10 sizes (N) in the range 1000-10000 and 6 densities (D=5%, 10%, 20%, 30%, 40% and 50%). We think that it is not necessary to process larger matrices, since we can use block decomposition methods for the DSMP were we reduce to submatrices (blocks) of lower sizes.

We experimented 8 versions of DNS, 7 of CSR, 7 of CSC and 3 of COO. We also studied the impact of loop unrolling [5], [9], [15], [17], when applied on DOT-R kernel for the two formats DNS and CSC. We present in the following excerpts of the results obtained for a density of 5% (similar results were obtained with the other values). We precise that our experiments have been achieved on an i7 work station (3.4 GHz, 4GB RAM, 64Ko L1 cache, 256 KO L2 cache and 8 MO

L3 cache)underopenSUSEOS.Ouralgorithms were coded in C.

A. *Intra-algorithm Comparison*

1) DNS

Despite its GAXPY-C kernel, JKI_V1 version offers minimal runtime (see Table I and Fig. 1). This is mainly due to the minimization of the number of logical tests (N^2) after loop invariant motion (LIM) from loop level 3 to loop level 2. Remark that for any density and any N, IKJ is about 62-85% better than IJK in average. Moreover, JKI_V1 is about 35-75% faster than JKI. In fact, this rate tends towards 32% when N and D increase. Thus we can conclude that the observed improvement is essentially due to LIM. Notice in addition that IKJ version whose kernel is GAXPY-R., is not among the best versions (it is ranked 4th to 6th in 60% of the tests) in spite of its efficient data locality (row-wise storing in a C environment, row-wise accesses).

2) CSR

The experimentations (see Fig. 2) confirm the theoretical results (see Table II). Indeed, IJK_V1 is the best since the GAXPY-R kernel reduced cache misses. Thus, we have data locality optimisation combined with code motion where we minimized access to IB i.e. 2N*NNZ to $2N^2$.We should however mention that the other versions, namely JIK_V1 and JKI_V1, uses less accesses to IB i.e. 2N but they are not the best since data locality has a more important impact that code motion in this case.

3) CSC

The best results were obtained with IJK_V1. This may be due to the reduced number of accesses to arrays JB and B as previously mentioned (see Table II). Moreover, matrices A and C are accessed row-wise (see Fig. 3).

4) COO

IK is the best since it adopts a row-wise access to the matrices i.e. GAXPY-R (see Table II and Fig. 4).

B. *Loop unrolling technique*

Given a (normalized) DO loop, loop unrolling (LpU) consists in first choosing an integer u (called LpU factor), duplicating the loop body u times, then iterating the loop with a step equal to u (instead of 1) [9]. It is well known that LpU reduces cache misses [5]. Thus, we applied it by choosing values for u in the range [2 32] and obtained interesting results. Indeed, the LpU technique has improved the algorithms on which it was applied (see Fig. 5 and 6). As a matter of fact, about an 8% improvement (in average) could be reached with u=4 for version COO_IK (GAXPY-R kernel, see Table III and Fig. 6). We precise that the parameter ratio used below is defined as follows :

*ratio=(1- run. time with unrolling /run. time without unrolling)*100*

Concerning DNS, for version DNS_IJK_V1 (DOT-R kernel), there is not a unique optimal LpU factor. As a matter of fact, in 33.33% of tests, optimality is obtained with u=16 (see Table VI and Fig. 5).

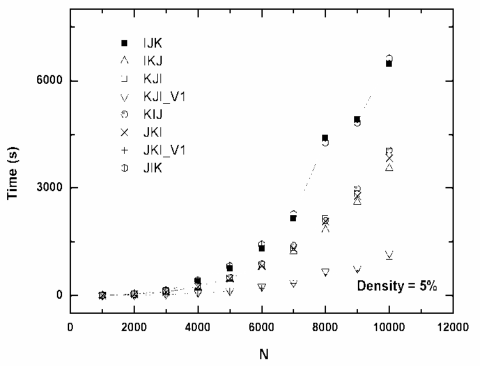

Fig. 1. Running times for DNS versions

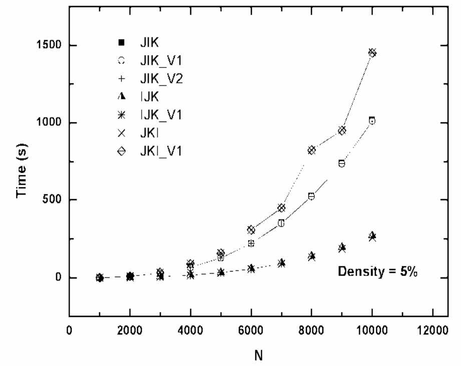

Fig. 2. Running times for CSR versions

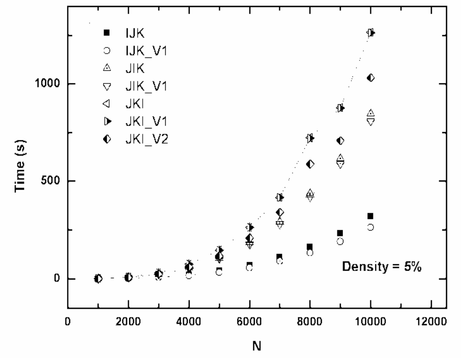

Fig. 3. Running times for CSC versions

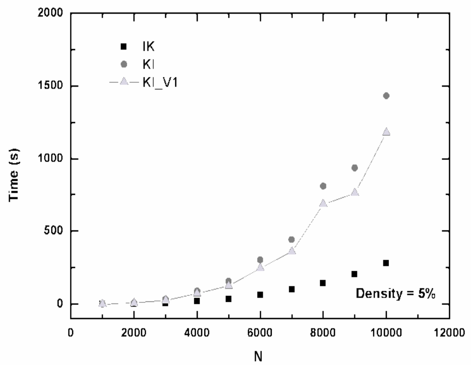

Fig. 4. Running times for COO versions

We can see that DNS_IJK_V1 version with unrolling which is more efficient than without (see Fig. 5) still remains less efficient than DNS_JKI_V1. Moreover, JKI_V1 is about 6 times faster than IJK_V1 with unrolling for D=5%. This improvement factor decreases in fact when D increases for fixed N (it reaches 1.5 for D=50% and any N).

Regarding CSR, for CSR_IJK_V1 (GAXPY-R kernel), the optimal unrolling factor is u=16 and induces, in average, a 10% improvement (see Table IV and Fig. 6).

As for CSC, CSC_IJK_V1 (DOT-R kernel) induced about a 43-46% improvement (in average). This was obtained with u=24 (see Table V and Fig. 6).

TABLE III. UNROLLING IMPROVEMENT RATIOS (%) FOR COO VERSIONS (DENSITY=5%)

N	u=2	u=4	u=8	u=12	u=16	u=20
1000	0.00	**0.00**	0.00	0.00	0.00	0.00
2000	0.00	**50.00**	50.00	50.00	50.00	50.00
3000	0.00	**14.29**	0.00	0.00	0.00	0.00
4000	5.56	**11.11**	11.11	5.56	11.11	11.11
5000	5.71	**8.57**	5.71	5.71	5.71	5.71
6000	4.92	**8.20**	6.56	6.56	6.56	6.56
7000	5.10	**9.18**	7.14	6.12	6.12	6.12
8000	6.21	**9.66**	7.59	6.90	6.21	6.21
9000	5.42	**8.37**	7.39	6.40	6.40	6.40
10000	5.32	**8.51**	7.45	6.38	6.03	6.03

TABLE IV. UNROLLING IMPROVEMENT RATIOS (%) FOR CSR VERSIONS (DENSITY=5%)

N	u=2	u=4	u=8	u=12	u=16	u=20
4000	-6.25	6.25	6.25	6.25	**6.25**	6.25
5000	-3.13	3.13	9.38	6.25	**9.38**	9.38
6000	-1.79	5.36	8.93	8.93	**8.93**	8.93
7000	-1.11	5.56	8.89	8.89	**10.00**	8.89
8000	-1.52	5.30	9.09	8.33	**9.85**	9.09
9000	-1.60	4.79	8.51	7.98	**9.57**	8.51
10000	-1.54	5.00	8.46	8.46	**9.62**	9.23

TABLE V. UNROLLING IMPROVEMENT RATIOS (%) FOR CSC VERSIONS (DENSITY=5%)

N	u=2	u=4	u=8	u=12	u=16	u=20	u=24	u=28
1000	10.20	30.61	42.86	38.78	35.71	34.69	**30.61**	24.49
2000	13.40	30.93	40.21	40.72	42.53	41.24	**40.89**	35.31
3000	13.06	30.63	39.64	40.54	42.91	42.43	**42.94**	39.26
4000	13.21	29.92	38.84	39.81	42.26	41.98	**42.30**	40.87
5000	12.50	29.77	38.61	39.87	41.95	41.79	**42.14**	41.07
6000	12.45	29.83	38.68	40.09	42.26	42.15	**42.53**	41.86
7000	12.14	29.69	38.61	39.89	42.70	42.91	**43.17**	42.77
8000	12.46	30.59	39.69	40.79	43.45	43.45	**43.82**	43.50
9000	12.62	30.70	39.85	40.95	43.52	43.58	**43.96**	43.81
10000	12.27	30.52	39.53	40.85	43.21	43.29	**43.68**	43.60

TABLE VI. UNROLLING IMPROVEMENT RATIOS (%) FOR DNS VERSIONS

N	D	u=4	u=8	u=12	u=16	u=20	u=24
1000	5%	-3.50	3.00	3.25	**4.00**	3.75	1.75
2000		5.52	12.73	7.52	**14.89**	14.61	13.59
3000		8.79	7.22	14.98	**15.60**	15.47	14.79
4000		13.05	13.89	14.44	**15.12**	14.90	14.12
5000		**15.03**	13.02	13.49	14.27	14.18	13.62
6000		9.48	12.54	12.96	**13.87**	13.36	13.14
7000		10.13	12.70	13.18	14.29	13.98	**15.70**
8000		36.34	37.68	37.92	**38.31**	38.06	37.67
9000		17.98	19.98	20.02	**20.81**	20.74	20.49
10000		14.72	16.38	16.73	**17.36**	17.14	16.64
1000	50%	6.45	5.55	**7.36**	6.55	5.45	2.55
2000		8.37	9.55	8.88	**10.20**	9.48	6.75
3000		30.96	29.29	**32.06**	31.44	30.73	28.40
4000		**10.81**	8.87	10.54	9.95	9.38	6.66
5000		**13.52**	7.90	9.75	9.05	8.38	5.61
6000		8.56	8.26	**10.00**	9.42	8.58	5.85
7000		8.11	7.60	**9.51**	8.62	8.11	5.87
8000		14.42	13.90	**15.64**	14.89	14.07	11.70
9000		9.09	8.54	**10.02**	9.32	8.81	6.04
10000		8.55	8.15	**9.80**	8.93	8.10	5.54

First Rang (%)	31.67	1.67	28.33	33.33	1.67	3.33

First Rank (%) above corresponds to the number of times (%) the corresponding version was first ranked. For instance, 33.33 in column u=16 means that for u=16, we obtained the best results in 33.33% of the 60 cases i.e. 20/60.

Concerning the behaviors of DNS and CSC algorithms according to the unrolling factor, we note that LpU technique significantly improved the performance but not so much as far as CSR and COO algorithms are concerned. In fact, DNS and CSC algorithms use DOT kernel while the two others use GAXPY kernel. When applying LpU technique with DOT kernel, the loop body corresponds to an accumulation in the same element of the matrix (see (h) below), whereas with GAXPY kernel, it is a set of assignments of different elements of the matrix (see (i) below).

CSC_IJK_u=4 ***DOT Kernel***

```
DO i=1,N
  DO j=1,N
     jbj= JB(j), jbj1= JB(j+1)-1
     m= (jbj1-jbj+1) mod 4 ; ne=jbj1-m ;
     DO k=jbj, ne,4
        C(i,j)=C(i,j)+A(i,IB(k))*B(k)+A(i,IB(k+1))*B(k+1)
               +A(i,IB(k+2))*B(k+2)+A(i,IB(k+3))*B(k+3)
     ENDDO
     DO k=ne+1, jbj1
        C(i,j)=C(i,j)+A(i,IB(k))*B(k)
     ENDDO
  ENDDO
ENDDO
```

(h)

```
COO_IK_u=4    GAXPY Kernel
 DO i=1,N
     m=NNZ mod 4 ; ne=NNZ-m
     DO k=1,ne,4
         C(i, JB(k))= C(i, JB(k))+A(i, IB(k))*B(k)
         C(i,JB(k+1))=C(i,JB(k+1))+A(i,IB(k+1))*B(k+1)
         C(i,JB(k+2))=C(i,JB(k+2))+A(i,IB(k+2))*B(k+2)
         C(i,JB(k+3))=C(i,JB(k+3))+A(i,IB(k+3))*B(k+3)
     ENDDO
     DO k=ne+1,NNZ
         C(i,JB(k))=C(i,JB(k))+A(i,IB(k))*B(k)
     ENDDO
 ENDDO
```

(i)

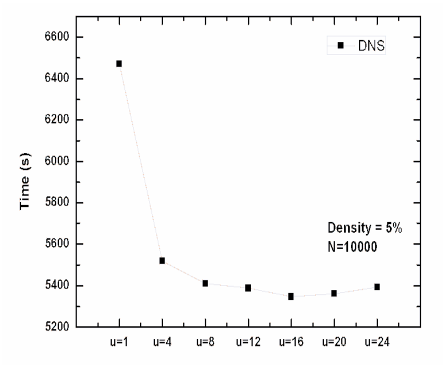

Fig. 5 . Running times for DNS with Loop unrolling

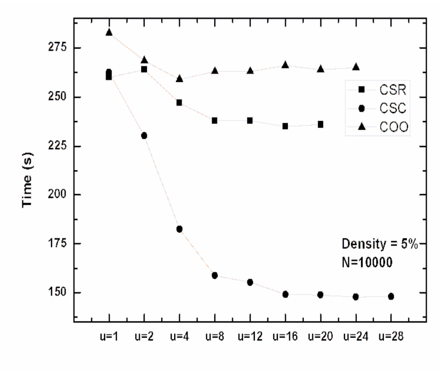

Fig. 6 . Running times for CSR, CSC and COO with Loop unrolling

C. *Inter-algorithms comparison*

1) Without unrolling

Before applying the LpU technique, two versions shared the first rank i.e. CSR-IJK_V1 (GAXPY-R kernel) and CSC-IJK_V1 (DOT-R kernel). Indeed, CSC-IJK_V1 (resp. CSR-IJK_V1) was the first in 55% (resp. 45%) cases. In fact, for low density (5%, 10%, 20%) CSR-IJK_V1 is the best whereas for higher densities (30%, 40%, 50%) CSC-IJK_V1 becomes better (see Table VII). Fig. 7 depicts the improvement ratios in terms of D for D=50% where ratio=(1-run_opt/run_v)*100, run_opt (resp. run_v) being the running time of version CSC_JIK_V1 (resp. the running time of any other version). A negative ratio means that run.time of CSC_JIK_V1 is larger than the other (i.e. CSR-IJK_V1 run.time is better).

We remark that CSC is followed by CSR, then COO and DNS. We can see that runtimes given by CSC version and CSR version are very close. In fact, in average, version CSC-IJK_V1 is 1-2% better than the best CSR-IJK_V1 (for high density), 6-12% better than the best COO version (i.e. COO-IK), and more than 50% better than the best DNS version i.e. DNS-JKI_V1 (see Fig. 8). Notice that the reduction of the number of logical tests (from N^3 to N^2) did not improve so much DNS-JKI_V1 since it was not associated to optimal data locality (row-wise).

TABLE VII. RATIO FOR THE OPTIMAL VERSION OF EACH FORMAT

N	D (%)	CSR_IJK_V1	COO_IK	DNS_JKI_V1
9000	5	-1.82	7.57	72.37
9000	10	-1.27	6.50	69.34
9000	20	0.04	8.01	63.43
9000	30	0.89	9.96	57.99
9000	40	1.50	10.58	52.92
9000	50	2.04	11.21	48.36
10000	5	-1.55	6.58	73.78
10000	10	-1.02	6.69	70.70
10000	20	-0.04	8.37	64.96
10000	30	0.74	9.86	59.15
10000	40	1.54	10.78	54.54
10000	50	1.97	11.39	50.26

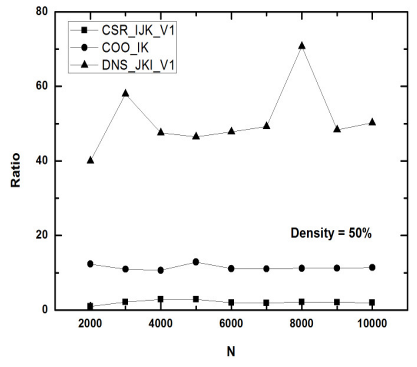

Fig. 7 . Ratio where D=50 % for the optimal version of each format

2) After unrolling

By applying the LpU technique, we can notice that COO_IK with unrolling factor u=4 is quite similar to CSR_IJK_V1 for low densities (D=5%). However, an 8-9% improvement could be obtained for u=16 by CSR_IJK_V1. On the other hand, CSC_IJK_V1_u=24 superseded CSR_IJK_V1_u=16.

Consequently, the ranking (best to worst) now is CSC_IJK_V1 (u=24), CSR_IJK_V1(u=16), COO_IK (u=4), DNS_JKI_V1. In fact, in average, CSC_IJK_V1 (u=24), is 34-37% better than CSR_IKJ_V1 (u=16), 36-42% better than COO_IK (u=4) and 80-85% (in average) better than DNS_JKI_V1 (see Fig. 8).

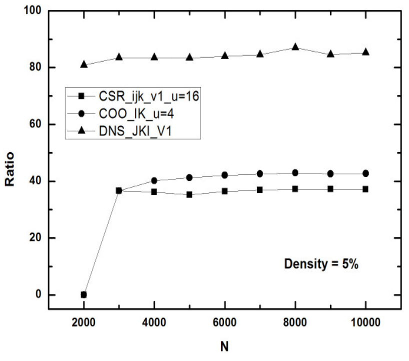

Fig. 8 . Ratio where D=5% for the optimal version

V. CONCLUSION

In this paper, we studied several dense-sparse matrix product algorithms for four compressing formats (DNS, CSR, CSC and COO). Various optimisation techniques have been applied and led to interesting improvements. The version corresponding to the CSC format, namely CSC_IJK_V1 (u=24), once optimised gave the best results and was followed by versions corresponding to successively formats CSR and COO, DNS.

Our work induces some interesting points we intend to study in the near future. We may cite the following :

- Extension of our work by studying other regular and irregular SCF's
- Comparison of SDMP and DSMP algorithms in order to deduce an adequate version for any kind of matrix product
- Study of the general case of sparse-sparse matrix product (SSMP)
- Study the sparse matrix chain product problem
- Parallelisation of SDMP and DSMP algorithms.

REFERENCES

[1] A. Buluç & J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication". In Proc of ICPP'08, pp. 503-510, Portland, Oregon, USA, 2008.

[2] T. A. Davis & Y. F. Hu, "The university of Florida sparse matrix collection", ACM Transactions on Mathematical Software, vol.38, N.1, pp. 1-25, 2011.

[3] E. Garcia, J. L. L. Pey, T. Juan, T. Lang & J. J. Navarro, "Block algorithms to speed up the sparse matrix by dense matrix multiplication on high performance worstations", Tech. Rep. No. UPC-DAC-1995-3, University Polytechnics of Catalunya, Barcelona, Spain, 1995.

[4] F. G. Gustavson, "Two fast algorithms for sparse matrices: multiplication and permuted transposition", ACM Transactions on Mathematical Software, vol. 4, N. 3, pp. 250-269, 1978.

[5] O. Hamdi-Larbi, N. Emad & Z. Mahjoub, "On sparse matrix-vector product optimization", In AICCSA'05, Cairo, Egypt, 2005.

[6] P. D. Sulatycke & K. Ghose, "Caching–efficient multithreaded fast multiplication of sparse matrices", In Proc. of the 12th. Int. parallel processing symposium on international parallel processing symposium, pp. 117-123, Orlando, FL, USA, 1998.

[7] Y. Saad, "Iterative methods for sparse linear systems", 2nd ed, SIAM Press, 2003.

[8] M. Bader & A. Heinecke, "Cache oblivious dense and sparse matrix multiplication based on Peano curves". In Proc. of the PARA 08, Lecture Notes in Computer Science, 6126/6127, 2008. From: https://para08.idi.ntnu.no/docs/submission_155.pdf

[9] S. Ezouaoui, Z. Mahjoub, L. Mendili & S. Selmi, "Performance evaluation of algorithms for sparse-dense matrix product", Proceedings of the International MultiConference of Engineers and Computer Scientists 2013, pp. 257-262, Kowloon, Hong Kong, 2013. From: http://www.iaeng.org/publication/IMECS2013/IMECS2013_pp257-262.pdf

[10] G. Greiner & R. Jacob, "The I/O complexity of sparse matrix dense matrix multiplication", LATIN 2010: Theoretical Informatics, in Lecture Notes in Computer Science 2010, 6034, pp. 143-156, Oaxaca, Mexico, 2010.

[11] G.W. Howell, "" Wide or Tall" and "Sparse matrix dense matrix" multiplications", In Proc. HPC '11, Proceedings of the 19th High Performance Computing Symposia, pp. 159-165, Boston, MA, USA, 2011.

[12] F. Ben Charrada, S. Ezouaoui, & Z. Mahjoub, "Greedy algorithms for optimal computing of matrix chain products involving square dense and triangular matrices", RAIRO - OR, vol. 45, N. 1, 1-16, 2011.

[13] E. Cohen, "Structure prediction and computation of sparse matrix products", Journal of Combinatorial Optimization, vol. 2, N. 4, pp. 307-332, 1998.

[14] I. S. Duff, A. M. Erisman & J. K. Reid, "Direct methods for sparse matrices", Oxford Science Publications, 1992.

[15] A. V. Aho, M. S. Lam, R. Sethi & J. D. Ullman, "Compilers: Principles, techniques, & tools", 2nd ed., Pearson Addison Wesley, 2007.

[16] V. Loechner, B. Meister & P. Clauss, "Precise data locality optimization of nested loops", The Journal of Supercomputing, vol. 21, N. 1, pp.37-76, 2002.

[17] J. J. Dongarra & A. R. Hinds, "Unrolling loops in FORTRAN", Software-Practice and Experience, vol. 9, N. 3, pp. 219-226, 1979.