Performance Evaluation of Algorithms for Sparse-Dense Matrix Product

S. Ezouaoui, Z. Mahjoub, L. Mendili and S. Selmi

Abstract—**We address in this paper the sparse-dense matrix product (SDMP) problem i.e. where the first (resp. second) matrix is sparse (resp. dense). 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. We then derive other versions by applying the techniques of loop interchange, loop invariant motion and loop unrolling. A theoretical multi-fold performance study permits to establish accurate comparisons between the different versions. Our contribution is validated through a series of experiments achieved on a set of sparse matrices of different sizes and densities. This leads to the choice of the best version, namely the one using COO format where a GAXPY-Row kernel combined with loop unrolling permitted to outperform the other versions by an improvement ratio of 20-30%.**

Index Terms— **Algorithm complexity, compressing/storing format, loop nest optimisation, performance evaluation, sparse matrix product**

I. Introduction

SPARSE computing corresponds to algorithms processing large sized sparse matrices. These latter are widely used in real world applications covering diverse domains such as electromagnetism, semi-conductors, robotics, image processing, networks and graphs, molecular dynamics, fluid dynamics, etc [1], [2], [3], [4], [5], [6]. The kernels used in these applications are mostly from sparse linear algebra [7]. The sparse-dense matrix product (SDMP) where the first is sparse and the second is dense is one among these frequent kernels, particularly used in iterative methods for solving linear systems or optimisation problems [4]. It is also used for building Peano space-filling curves [8], data completion in inverse problems based on the Steklov-Poincaré discrete approach [9] which uses matrix conditioning [10], Krylov and block Lanczos methods [11].

Several works have been devoted to the SDMP problem [8], [11], [12]. But we have to mention some points that remain not studied as far as we know. As a matter of fact, in [12], the authors designed theoretical algorithms but did not take into account the important point of the matrix access/storing modes and data locality. Others [4], [8] did not study the impact of sparse storing formats.

It is noteworthy that optimising loop nest structured algorithms such as the SDMP one particularly requires avoiding useless operations, improving data locality optimising memory accesses and reducing cache missing.

On the other hand, we have to underline that processing large sparse murices requires, for reasons of space-time complexity reduction, the use of compressing (or storing) formats (SCF). These latter may be either general i.e. adapted to any sparse structure e.g. DNS (DeNSe), 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 the determination of the best SCF for the SDMP i.e. leading to the best performances. For this purpose, starting from original algorithms, we derived a series of others corresponding to four SCF's, namely DNS, CSR, CSC and COO.

The remainder of the paper is organised as follows. In section II, we present and compare several algorithms for the SDMP corresponding to the four chosen SCF's. Section III is devoted to an experimental study in order to validate our theoretical contribution.

II. Theoretical Study

Let us recall that a matrix is called *sparse* if it has a large (resp. weak) number of zero (resp. non zero) elements [5]. Let NNZ be the number of nonzero elements. As previously mentioned, processing sparse matrices requires using particular SCF's restricted to the nonzero elements. The study of the SDMP is in fact based on the Sparse Matrix Vector Product (SMVP) where the matrix is sparse and the vector is dense. It is well known that, for a matrix of size N, the SMVP requires $2*NNZ$ flops [5]. Therefore, the SDMP will require $2N*NNZ$ flops.

A. DNS versions

The standard algorithm (for dense matrix product C=AB) has the structure of three perfectly nested loops, denoted IJK (see version (a) below) and is of cubic complexity. From this original version, we include both logical tests and scalar replacements in order to avoid useless operations and reduce the number of accesses to matrix A. Obviously, five other versions may be derived by means of the loop interchange (LI) technique i.e. IKJ, KJI, KIJ, JKI, JIK [4], [6]. Some of these lead to both better data locality and computing kernel (see Table I, where R is for Row and C for Column) [4], [6]. As will be seen below, in addition to LI, another loop nest optimisation will be introduced.

S. Ezouaoui, is a doctoral student at the University of Tunis El Manar, Faculty of Sciences of Tunis, University Campus - 2092 Manar II – Tunis, Tunisia (phone: ++21696276502; e-mail: zouaoui.sana29@gmail.com).

Z. Mahjoub is Professor of Computer Science at the University of Tunis El Manar, Faculty of Sciences of Tunis, University Campus - 2092 Manar II – Tunis, Tunisia (e-mail: zaher.mahjoub@fst.rnu.tn).

L. Mendili and S. Selmi have obtained in 2010 their B.S degree in CS at the University of Tunis El Manar, Faculty of Sciences of Tunis, University Campus - 2092 Manar II – Tunis, Tunisia, (e-mails: Mendili.hiba@gmail.com ; amatoallah44@hotmail.fr)

ISBN: 978-988-19251-8-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Keeping the IKJ version having the best kernel i.e. GAXPY-R (in a C programming environment, see (b) below), we can apply the loop invariant motion (LIM) technique [13] (see (c) below) in order to reduce the number of logical tests. It is known that LIM is an efficient code optimisation procedure for loop nests. In our case, it consists in moving to the second level the instruction's=A(i,k)' and the following IF test. Table I recapitulates a comparative study on the whole 6 versions as far as the number of accesses, number of tests, access mode and kernel are concerned.

DNS_IJK

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

(a)

DNS_IKJ

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

(b)

DNS_IKJ_V1

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

(c)

We thus notice, from Table I, that versions V1 of KIJ and IKJ lead to the best results (number of accesses and logical tests) i.e. N^2 (resp. NNZ) accesses to A (resp. B) instead of N^3 (resp. N*NNZ) for the 4 other versions.

Let us add that the kernel used by both IKJ and JKI is GAXPY (same access mode for the three matrices). Therefore, a better data locality leading to less cache misses. Indeed, in order to reduce these latter, data must be processed according to their storing mode (i.e. row-wise or column-wise). We recall that in C (resp. Fortran), arrays are stored row-wise (resp. column-wise). Thus, we have to keep the version suited to the programming environment and leading to the best performances [14].

B. CSR, CSC and COO versions

The design of the SDMP algorithm where A is stored either in CSR, CSC or COO format, may be easily derived from the corresponding SMVP algorithm [4]. Hence, we get a first version for each SCF i.e. CSR_JIK, CSC_JIK and COO_JI. We then apply improving techniques such as scalar replacement and loop interchange in order to have better data locality (see versions (e), (f) and (g) below).

Using the LI technique, two other versions i.e. IJK and IKJ were directly derived from the initial CSR_JIK and CSC_JIK. As for COO, we derived a new version i.e. IJ from the initial JI (see Part I in Table II).

Let us precise that, due to the non affine loop bounds (i.e. array elements) of loop K in the JIK versions of both CSR and CSC, versions KIJ and KJI cannot be directly derived. Indeed, we first transformed the K loop bounds into affine (in fact constant) ones through the use of an IF test instruction (see (e-1)). This could be done since we know that array IA in CSR_JIK (resp. JA in CSC_JIK) have NNZ elements. The IF test instruction will restrict iterations only to the right ones (see version (e-1) below).

The derived versions, namely KJI, KIJ, JKI of CSR and KJI, KIJ, JKI of CSC are recapitulated in Table II (see Part II).

CSR_JIK_V1

```
DO j=1, N
  DO i=1, N
    iai=IA(i) ; iai1=IA(i+1)-1
    DO k=iai,iai1
      C(i,j)=(i,j)+A(k)*B(JA(k),j)
    ENDDO
  ENDDO
ENDDO
```

(e)

CSR_JIK intermediate version

```
DO j=1, N
  DO i=1, N
    DO k=1, NNZ
      IF (IA(i) ≤k≤ IA(i+1)-1) THEN
        C(i,j)=C(i,j)+A(k)*B(JA(k),j)
      ENDIF
    ENDDO
  ENDDO
ENDDO
```

(e-1)

CSR_KIJ_V1

```
DO k=1, NNZ
  ak= A(k); jak= JA(k)
  DO i=1,N
    IF (IA(i) ≤k≤ IA(i+1)-1) THEN
      DO j=1,N
        C(i,j)=C(i,j)+ak*B(jak,j)
      ENDDO
    ENDIF
  ENDDO
ENDDO
```

(e-2)

ISBN: 978-988-19251-8-3 (revised on 18 March 2013)
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

CSC_IKJ_V1

```
DO i=1, N
    jai=JA(i) ;  jai1=JA(i+1)-1
    DO k= jai, jai1
        iak=IA(k) ;  ak=A(k)
        DO j=1,N
            C(iak,j)= C(iak,j)+ak*B(i,j)
        ENDDO
    ENDDO
ENDDO
```

(f)

COO_IJ_V1

```
DO i=1, NNZ
    iai=IA(i); jai=JA(i)
    DO j=1, N
        C(iai,j)=C(iai,j)+A(i) *B(jai,j)
    ENDDO
ENDDO
```

(g)

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	# Test	# Access	# Test		# Access	# Access			
IJK	N^3	N^3	N^3	N^3	R	N*NNZ	N*NNZ	C	R	DOT-R
JIK	N^3	N^3	N^3	N^3	R	N*NNZ	N*NNZ	C	C	DOT-C
KIJ	N^3	N^3	$\mathbf{N^2}$	$\mathbf{N^2}$	C	N*NNZ	**NNZ**	R	R	AXPY-R
KJI	N^3	N^3	N^3	N^3	C	N*NNZ	N*NNZ	R	C	AXPY-C
IKJ	$\mathbf{N^3}$	$\mathbf{N^3}$	$\mathbf{N^2}$	$\mathbf{N^2}$	**R**	**N*NNZ**	**NNZ**	**R**	**R**	**GAXPY-R**
JKI	N^3	N^3	N^3	N^3	C	N*NNZ	N*NNZ	C	C	GAXPY-C

TABLE II
SCF'S COMPARISION

SCF		Version	Initial version (# access)			V_1 (# access)			A	B	C	Kernel
			IA	JA	A	IA	JA	A	AM	AM	AM	
CSR	Part I	**JIK**	2N*NNZ	N*NNZ	N*NNZ	$2N^2$	N*NNZ	N*NNZ	R	C	C	DOT-C
		IJK	2N*NNZ	N*NNZ	N*NNZ	2N	N*NNZ	N*NNZ	R	C	R	DOT-R
		IKJ	2N*NNZ	N*NNZ	N*NNZ	2N	N*NNZ	N*NNZ	R	R	R	GAXPY-R
	Part II	**JKI**	$2N^2$*NNZ	N^2*NNZ	N^2*NNZ	$2N^2$*NNZ	N*NNZ	N*NNZ	C	C	C	GAXPY-C
		KJI	$2N^2$*NNZ	N^2*NNZ	N^2*NNZ	$2N^2$*NNZ	NNZ	NNZ	C	R	C	AXPY-C
		KIJ	$2N^2$*NNZ	N^2*NNZ	N^2*NNZ	2N*NNZ	NNZ	NNZ	C	R	R	AXPY-R
CSC	Part I	**JIK**	2N*NNZ	N*NNZ	N*NNZ	N*NNZ	$2N^2$	N*NNZ	C	C	C	GAXPY-C
		IJK	2N*NNZ	N*NNZ	N*NNZ	N*NNZ	2N	N*NNZ	C	R	C	AXPY-C
		IKJ	2N*NNZ	N*NNZ	N*NNZ	NNZ	2N	NNZ	C	R	R	AXPY-R
	Part II	**JKI**	$2N^2$*NNZ	$2N^2$*NNZ	N^2*NNZ	N*NNZ	$2N^2$*NNZ	N*NNZ	R	C	C	DOT-C
		KJI	$2N^2$*NNZ	$2N^2$*NNZ	N^2*NNZ	NNZ	$2N^2$*NNZ	NNZ	R	C	R	DOT-R
		KIJ	$2N^2$*NNZ	$2N^2$*NNZ	N^2*NNZ	NNZ	2N*NNZ	NNZ	R	R	R	GAXPY-R
COO		**JI**	2N*NNZ	N*NNZ	N*NNZ	N*NNZ	N*NNZ	N*NNZ	R	C	C	DOT-C
		IJ	2N*NNZ	N*NNZ	N*NNZ	N	N	N*NNZ	R	R	R	GAXPY-R

AM : Access mode (Row/Column) ; Part I : first 3 directly derived versions ; Part II : second three indirectly derived versions (loop restructuring, body modification)

TABLE III
INDIRECTLY DERIVED VERSIONS AND CORRESPONDING COO VERSIONS

SCF	Version	A	B	C	Kernel	COO
		AM	AM	AM		
CSR	**JKI_JK**	R	C	C	DOT-C	JI
	KJI_KJ	R	R	R	GAXPY-R	IJ
	KIJ_KJ	R	R	R	GAXPY-R	IJ
CSC	**JKI - JK**	C	C	C	GAXPY-C	--
	KJI_KJ	R	R	R	GAXPY-R	IJ
	KIJ -KJ	R	R	R	GAXPY-R	IJ

ISBN: 978-988-19251-8-3 (revised on 18 March 2013)
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Starting from versions KIJ for both CSR and CSC, we may apply the loop invariant motion technique and move the logical test to an upper level, thus reducing accesses. This technique applied for versions KIJ and KJI permits to reduce accesses to arrays A and JA in CSR versions (resp. arrays A and IA in CSC versions).

Hence, from three initial versions (see Part I of Table II), we derived 16 other versions i.e. 5 for CSR, 8 for CSC and 3 for COO, so a total of 19 (see Table II). Notice that the number of accesses to array B always remains the same (i.e. N*NNZ). Table II recapitulates the results obtained for the three formats. It is easy to remark that through comparative intra-SCF versions, CSR_IKJ_V1, CSC_IKJ_V1 and COO_IJ_V1 are (theoretically) the best when coded in C language as we did (row-wise storing). Indeed, they perform firstly 2 to 3 (out of 3) row accesses to A, B, and C and secondly less accesses to these arrays. We precise that in the COO format, we choosed a row-wise storing for A.

As to the second group of versions (see Part II in Table II), we could derive 3 versions for CSR as well as for CSC. Notice here that loop invariant optimisation permitted to reduce the number of accesses to arrays JA and A (resp. IA and A) in CSR (resp. CSC) versions i.e. from N^2*NNZ to NNZ (see CSR version (e-2) above). As to array IA in CSR-KIJ (resp. JA in CSC-KIJ) version, we reduce from $2N^2*NNZ$ to $2N*NNZ$.

Remark on the other hand, that those three versions lead for CSR to kernels GAXPY-C (JKI), AXPY-R (KIJ) and AXPY-C (KJI) which is not interesting, since we intend to use in our experiments a C environment for which the GAXPY-R kernel, already obtained with CSR-IKJ, is the best. We have to finally say that, in the three directly derived CSR versions, the numbers of accesses to arrays A, B and C are fewer than in the last three versions.

As to the last three versions for CSC, they led to kernels DOT-R (KJI), DOT-C (JKI) and GAXPY-R (KIJ), the last being the best kernel in a C environment.

C. Relations between SCF's and derived versions

If we consider any version among the 3 last versions (indirectly) derived for CSR and CSC, we can easily remark that a restructuring permits to eliminate a loop (among the three) namely the I loop scanning the rows (resp. columns) of array A in CSR (resp. CSC) versions. In fact, this elimination procedure requires the creation of a new list, denoted ROW, of size NNZ and involving, in CSR (resp. CSC) versions, the row (resp. column) indices of array A corresponding to the row indices of array C (resp. B). Table III recapitulates these facts (see algorithms (h) and (i) below).

We have to notice that these 2-loop CSR and CSC versions are identical to COO versions (see previous section, algorithm (g)). Indeed, from Table III, we see that (i) CSR_JKI_JK is equivalent to COO-JI, (ii) CSR_KJI_KJ, CSR_KIJ_KJ, CSC_KJI_KJ and CSC_KJI_KJ are equivalent to COO_IJ. However, CSC_JKI_JK have no corresponding. As a matter of fact, COO_JI has a GAXPY-R kernel, since array A elements are stored row-wise whereas in CSC_JKI_JK, we have a GAXPY-C kernel.

CSR_JKI _JK

```
DO j=1, N
    DO k=1, NNZ
        C(ROW(k),j)=C(ROW(k),j)+A(k)*B(JA(k),j)
    ENDDO
ENDDO
```

(h)

CSC_JKI_JK

```
DO j=1,N
    DO k=1, NNZ
        C(IA(k),j)=C(IA(k),j)+A(k)*B(ROW(k),j)
    ENDDO
ENDDO
```

(i)

III. Experimental Study

A series of experimentations have been achieved in order to evaluate the practical performances of the derived versions and validate our theoretical study. For this purpose, we have chosen 10 matrix sizes (N) in the range 1000-10000 and 6 densities (D=5%, 10%, 20%, 30%, 40% and 50%). The matrices were randomly generated. Remark that it seems useless to process larger matrices, since we can use block methods for the SDMP were we reduce to blocks of lower sizes. We therefore experimented 8 versions of DNS, 6 of CSR, 9 of CSC and 4 of COO. Let us add that we also studied the impact of loop unrolling [5], [13], [15] when applied to the kernel DOT-R for the three formats DNS, CSR et COO. As to the CSC format, its DOT kernel is not interesting since the number of array accesses is quite large.
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 we used an i7 work station (3.4 GHz clock, 4GB RAM, 64Ko L1 cache, 256 KO L2 cache and 8 MO L3 cache) under openSUSE OS. Our algorithms were coded in C.

A. Intra-algorithm Comparison

- DNS : the best running times were obtained with IKJ_V1 (see Table I and Fig.1). This is due to the following reasons: (i) data locality optimisation (row-wise storing in a C environment, row-wise accesses), (ii) minimisation of the number of logical tests (N^2) after loop invariant motion (LIM) from level 3 to level 2. Remark that for d=5 and any N, IKJ is about 20% better than IJK. This ratio increases with d (for any N) and reaches 50% in average. Moreover, IKJ_V1 is about 15 times faster than IKJ for d=5. This factor naturally decreases with d for fixed N (reaches 2 for d=50 and any N). Thus we can conclude that the improvement is essentially due to LIM.
- CSR : The experimentations (see Fig. 2) confirm the theoretical results (see Table II). Indeed, IKJ_V1 is the best since the GAXPY-R kernel reduced cache misses.
- CSC : The best results are obtained with IKJ_V1. This may be justified by the reduced number of accesses to arrays JA and A as previously mentioned (see Table II). Moreover, matrices B and C are accessed row-wise (see Fig.3).
- COO : IJ_V1 is the best since it firstly adopts a row-wise access to the matrices i.e. GAXPY-R (see Table II) and secondly performs a reduced number of accesses to arrays IA and JA (see Fig. 4).

ISBN: 978-988-19251-8-3 (revised on 18 March 2013)
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

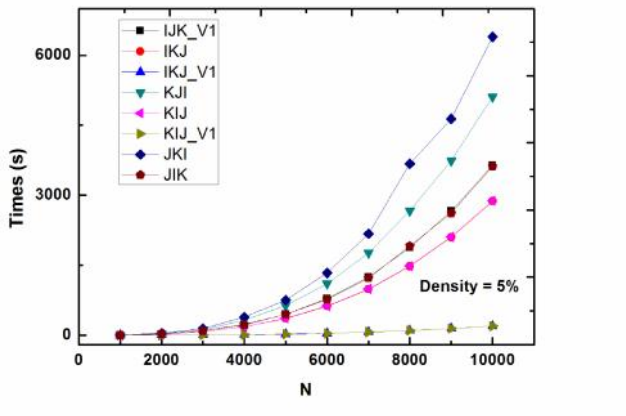

Fig. 1 Running times for DNS versions

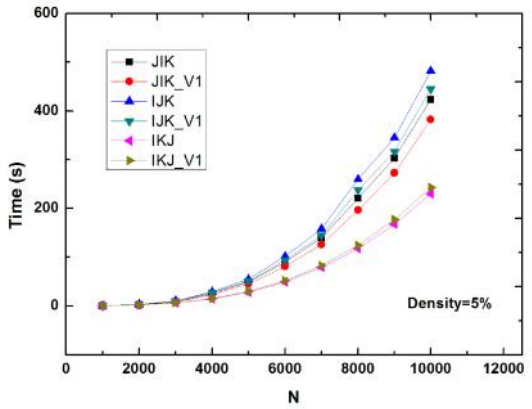

Fig. 2 Running times for CSR versions

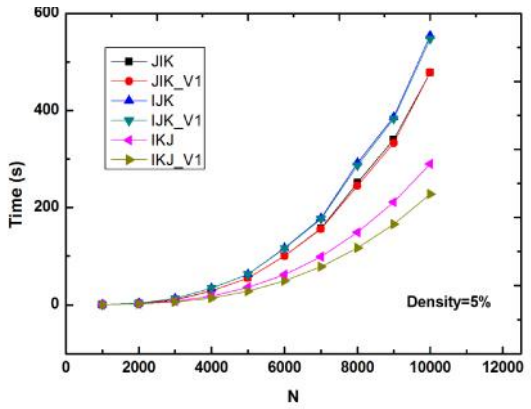

Fig. 3 Running times for CSC versions

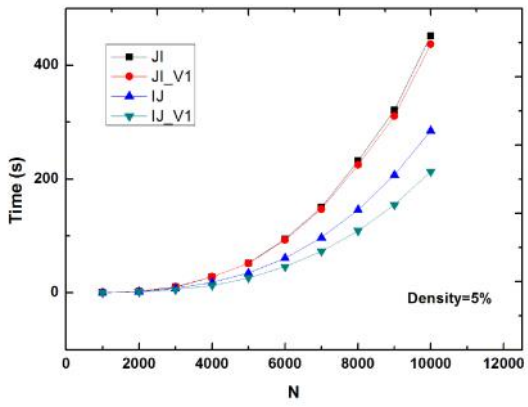

Fig. 4 Running times for COO versions

B. *Loop unrolling technique*

Given a (normalised) DO loop, loop unrolling (LU) consists in first choosing an integer u (named LU factor), duplicating the loop body u times, then iterating the loop with a step equal to u (instead of 1). It is well known that LU reduces cache misses [5]. Thus, we applied LU by choosing values for u in [2 20] and obtained interesting results. Indeed, about a 44% improvement (in average) could be reached with u=12 for version DNS_IJK_V1 (see Table VI). As to version COO_IJ_V1 (GAXPY-R kernel), a 24-25% improvement could be obtained (u=16). However, for version COO_JI_V1 (DOT kernel), an improvement of only 1% was reached (u=4, see Table IV). Concerning CSR, for version CSR_IJK_V1 (DOT-R kernel), we reached an improvement of 27-38% (u=12).

TABLE VI
UNROLLING IMPROVEMENT RATIOS (%) FOR DNS, CSR AND COO VERSIONS (DENSITY=5%)

	N	u=2	u=4	u=8	u=12	u=16
DNS_IJK	8000	11.24	21.19	23.88	**43.64**	25.83
	9000	10.76	20.94	23.91	**43.87**	25.94
	10000	10.39	20.51	23.80	**43.78**	25.79
	N	**u=2**	**u=6**	**u=8**	**u=12**	**u=16**
CSR_IJK	8000	19,92	30,67	31,26	**31,93**	31,93
	9000	18,61	28,08	29,10	**29,65**	29,65
	10000	16,63	25,84	26,69	**27,19**	27,19
	N	**u=2**	**u=4**	**u=8**	**u=16**	**u=20**
COO_IJ	8000	15.60	21.10	23.85	**25.08**	7.34
	9000	15.27	20.86	23.87	**24.73**	6.67
	10000	15.26	20.89	23.94	**25.12**	7.04
	N	**u=2**	**u=4**	**u=8**	**u=12**	**u=16**
COO_JI	8000	0.87	**1.48**	-11.52	-10.20	-24.20
	9000	0.42	**1.90**	-8.15	-11.53	-26.77
	10000	0.23	**1.35**	-15.80	-11.74	-24.94

The ratio is defined as follows:
*ratio=(1- run. time with unrolling /run. time without unrolling)*100*
A negative ratio means that run time with unrolling is larger than

C. *Inter-algorithms comparison*

– **Before** applying the LU technique, version DNS-IKJ_V1 is the best among all. Fig. 5 depicts the improvement ratios in terms of d for N=10000. The ratio is defined by *ratio=(1-run_opt/run_v)*100* where *run_opt* is the running time of version DNS_IKJ_V1 and *run_v* is the running time of any other version. We hence remark that DNS is followed by COO, then CSC and CSR. In fact, in average, version DNS_IKJ_V1 is 7% better than the best COO_JI_V1, 12-14% better than the best CSC version i.e. CSC_IKJ_V1, and 16-18% better than the best CSR version i.e. CSR_IKJ_V1 (see Fig. 5). Notice that the dramatic reduction of the number of logical tests (from N^3 to N^2) associated to an efficient data locality and direct accesses are the main reasons that 'boosted' version DNS_IKJ_V1. Let us add that even if version CSR_IKJ_V1 has a GAXPY_R kernel, it is outperformed by version CSC_IKJ_V1 which has an AXPY_R kernel for the latter has less accesses to array A, IA and JA (see Table II).

ISBN: 978-988-19251-8-3 (revised on 18 March 2013) IMECS 2013
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Remark that If no LIM nor scalar replacement is done, version DNS_IKJ would be the worst and we would have the following ranking : CSR_IKJ, COO_IJ, CSC_IKJ, DNS_IKJ. Notice in addition that version CSC_IKJ has an AXPY-R kernel whereas both versions CSR_IKJ and COO_IJ have a GAXPY-R kernel, thus better access mode and data locality (see Table V, excerpts for d=5%).

- **After** applying the LU technique, we have a new ranking i.e. COO_IJ_V1 (u=16), DNS_IKJ_V1, CSC_IKJ_V1, CSR_IKJ_V1. In fact, in average, COO_IJ_V1 (u=16), is 19-20% better than DNS_IKJ_V1, 30% better than CSC_IKJ_V1 and 32-33% better than CSR_IKJ_V1 (see Fig. 6).

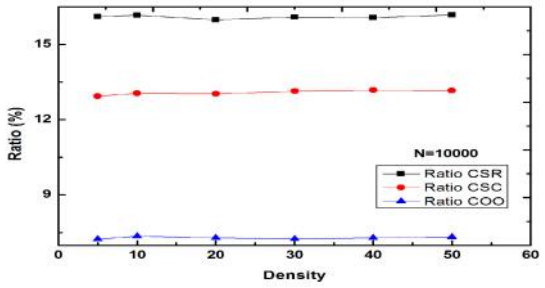

Fig. 5 Ratio in terms of density for the optimal version of each format

TABLE V
TIME (S) FOR DNS, CSR AND COO VERSIONS WITHOUT OPTIMIZATION (D=5)

N	CSR_IKJ	COO_IJ	CSC_IKJ	DNS_IKJ
5000	28	35	36	297
6000	49	61	62	513
7000	79	97	99	814
8000	118	146	149	1215
9000	168	207	211	1728
10000	230	285	290	2370

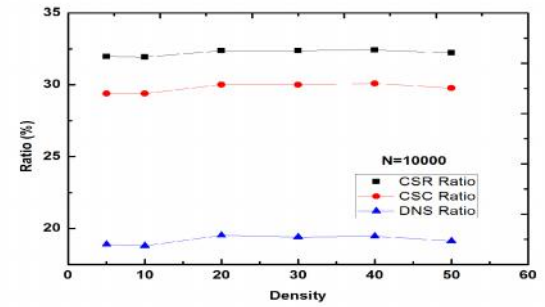

Fig. 6 Ratio in terms of density for the optimal version of each format (after unrolling)

IV. CONCLUSION

We studied in this paper, diverse algorithms for the sparse dense matrix product associated to four sparse compressing formats. Efficient optimisation techniques have been applied and led to interesting improvements. The version corresponding to the COO format (i.e. where the sparse matrix is stored in three list arrays of size NNZ), namely COO_IJ_V1 (u=16), once optimised gave the best results and was followed by versions corresponding to the formats DNS, CSC and CSR. If we exclude DNS, the COO leader version is about 30% better than the versions corresponding to the two others. However, the space complexity of the COO format requires NNZ-N more integers than the others.

Another result to underline is that the technique of loop interchange (LI) permitted to discover a particular relation between the CSR, CSC and COO formats versions. Indeed, LI applied to CSR and CSC versions led to COO versions.

Finally, our work induces some interesting points, we intend to study in the near future. We may cite (i) the dense-sparse matrix product (DSMP) problem ; (ii) the general case of the sparse-sparse matrix product problem (SSMP) ; (iii) the sparse matrix chain product problem [16], [17] ; (iv) the parallelisation of SDMP algorithms.

ACKNOWLEDGMENT

Special thanks are addressed to Dr. O. Hamdi-Larbi for her valuable help.

REFERENCES

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

[2] T. A. Davis & Y. F. Hu, "The university of Florida sparse matrix collection", *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, Nov. 2011.

[3] F. G. Gustavson, "Two fast algorithms for sparse matrices: multiplication and permuted transposition", *ACM Transactions on Mathematical Software*, vol. 4 no. 3, pp. 250–269, Sep.1978.

[4] E. Garcia, J. L. L. Pey, T.Juan, T.Lang and J. J. Navarro, "Block algorithms to speed up the sparse matrix by dense matrix multiplication on high performance worstations", MA Rep., UPC-DAC-1995-3, University Polytechnics of Catalunya, Barcelona, Spain ,1995.

[5] O. Hamdi-Larbi, N. Emad and Z. Mahjoub, "On sparse matrix-vector product optimization, In *Proc. AICCSA '05*, Cairo, Egypt, 2005, pp.23.

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

[7] Y.Saad, "*Iterative methods for sparse linear systems*", 2nd ed, SIAM Press, 2003, pp 92–95, 380 –385.

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

[9] M. Azaiez, F. Ben Belgacem and H. El Fekih, "On Cauchy's problem II: completion, regularization and approximation", *Inverse Problems,* vol. 22, no. 4, pp. 1307–1336, Aug.2006.

[10] R. Ben Fatma, "Completion data for the Helmholtz equation: Application to some problems Inverse", Ph.D. dissertation, Dept. Math, National school of engineering of Tunis, Tunisia, 2012.

[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*, 2011, pp. 159−165.

[12] G.Greiner and R. Jacob, "The I/O Complexity of Sparse Matrix Dense Matrix Multiplication", LATIN 2010: Theoretical Informatics, in *Lecture Notes in Computer Science 2010*, Vol. 6034, 143−156.

[13] A. V. Aho, M. S.Lam, R.Sethi and J. D.Ullman, "Compilers: Principles, Techniques, & Tools". 2nd edi, Pearson Addison Wesley, 2007, pp. 592, 738.

[14] V. Loechner, B. Meister and P. Clauss, "Precise Data Locality Optimization of Nested Loops", *The Journal of Supercomputing*, vol. 21, no. 1, pp. 37−76, Jan. 2002.

[15] J. J. Dongarra and A. R. Hinds, "Unrolling Loops in Fortran", *Software-Practice and Experience*, vol. 9, no. 3, pp. 219−226, Mar. 1979.

[16] E. Cohen, (1998, December), "Structure prediction and computation of sparse matrix products", *Journal of Combinatorial Optimization*, vol. 2, no. 4, pp. 307–332, Mar. 1979.

[17] F. Ben Charrada, S. Ezouaoui and Z. Mahjoub, "Greedy algorithms for optimal computing of matrix chain products involving square dense and triangular matrices", *RAIRO - OR*, vol. 45, no.1, pp. 1−16, Jan. 2011.

This paper was changed at 3/3/2013. Modifications are made in Table VI page 5 and in the unrolling improvement of CSR algorithm (between 27-38%).