Efficiently parallel implementation of the inverse Sherman-Morrison algorithm

Xin He, Marcus Holm and Maya Neytcheva

Department of Information Technology, Uppsala University, Sweden
he.xin,marcus.holm,maya.neytcheva@it.uu.se

Abstract. We contribute two parallel strategies to compute the exact and approximate inverse of a dense matrix, based on the so-called inverse Sherman-Morrison algorithm and demonstrate their efficiencies on multicore CPU and GPU-equipped computers. Our methods are shown to be much better than a common matrix inverse method, yielding up to 12 times faster performance. A comparison of the performance of the CPU and GPU versions is made and analyzed with the aid of a performance model.

1 Introduction

The task to compute explicitly the exact inverse of a given nonsingular matrix is among the heaviest computational kernels in matrix linear algebra. We consider a real square nonsingular matrix $A$ of size $n \times n$ and pose the task to compute $A^{-1}$. We restrict ourselves to the class of dense matrices.

First, we briefly mention some well-known algorithms, and then we present our contribution. One straightforward approach to compute $A^{-1}$ could be to determine its columns as solutions of the linear systems $LUx_i = I_n$, where $LU$ is the exact factorization of $A$ and $I_n$ is the identity matrix of size $n$. The so-obtained matrix $X = \{x_i\}$, $i = 1, \cdots n$ is then the inverse of $A$. As is well-known, the computational cost to factorize $A$ is $O(n^3)$ and each solution with $L$ and $U$ costs $O(n^2)$ operations. The total cost to compute $A^{-1}$ is then $O(n^3)$.

Another often used method to compute $A^{-1}$ is via the Gauss-Jordan method. Its computational cost is also $O(n^3)$. Both of these algorithms are numerically unstable and require permutations.

For sufficiently large $n$, it becomes critical to achieve full utilization of complex computer hardware resources. It is well-known that, for instance, the solution of systems with triangular matrices is inherently serial and it is, therefore, not likely that such an algorithm could be efficiently implemented on multicore and GPU-equipped computers. Applying permutations is another matrix manipulation, which is not easily parallelizable.

Instead, we consider here an approach based on the so-called inverse Sherman-Morrison (ISM) algorithm, explained below. The major difference is that although the computational complexity of the ISM algorithm is still $O(n^3)$, it can be written in a block form that with only BLAS3 operations. Provided that we possess highly efficient BLAS operations, tuned for multicore architectures or GPUs, we can expect that for large enough problems, the ISM implementation would use all the computational resources very efficiently and, thus, could outperform other methods, such as the above mentioned ones.

To apply ISM, we need to represent $A$ in a special form, namely, as

$$A = A_0 + XY^T$$

(1)

where $A_0 \in \mathbb{R}^{n \times n}$ is a matrix, whose inverse is easy to compute (e.g., $A_0$ could be diagonal, or even the identity matrix) and $X, Y \in \mathbb{R}^{n \times m}$.

We note first, that any matrix can be represented in the form (1), for instance by taking $A_0$ to be the diagonal of $A$, $X = A - A_0$ and $Y = I_n$. Clearly, the representation of $A$ in the form (1) is not unique. There are numerous application areas, however, where we need to compute the exact inverses of matrices, which arise directly in the form (1), as in some statistical problems, for example, related to seismic, genetic studies, in certain flow problems etc. Moreover, due to the
treatment of large data sets, \( n \) can often be of order \( 10^6 \) or more, and \( m \) can be of order \( 10^4 \) to \( 10^6 \). In some applications \( X \) and \( Y \) can be sparse, however taking advantage of the sparsity falls out of the scope of this work.

In certain cases, such as in some statistical applications, the exact inverse of \( A \) has to be computed explicitly and is needed in further analysis. In other cases, it suffices to compute only an approximation of \( A^{-1} \), to be used as a high quality multiplicative preconditioner to \( A \), when applying iterative solution methods in large scale scientific computations. Dropping relatively small-valued entries, related to some given tolerance, is the usual technique to obtain a sparse approximation of \( A^{-1} \), however this is also left out of the scope of this paper.

The ISM algorithm and a block version of it have been derived and studied earlier in [1,2,3]. We focus here on the parallel implementation of the ISM algorithm on two computer architectures - shared memory multicore CPU and GPU. The paper is organized as follows. The recursive (referred here to as the single vector) form and the two block forms of the ISM algorithm are presented in Section 2. Performance results for CPU and GPU implementations are reported in Section 4 and a discussion can be found in Section 5.

2 The ISM algorithm

2.1 The single vector ISM algorithm

Let \( A \) be of the form (1) and \( I_m \in \mathbb{R}^{m \times m} \) be the identity matrix of size \( m \). The Sherman-Morrison-Woodbury formula provides an explicit form of \((A_0 + XY^T)^{-1}\), given by the expression

\[
(A_0 + XY^T)^{-1} = A_0^{-1} - A_0^{-1}X(I_m + Y^T A_0^{-1} X)^{-1} Y^T A_0^{-1}, 
\]

(2)

provided that the matrix \( I_m + Y^T A_0^{-1} X \) is nonsingular.

Applying formula (2) on the columns of \( X \) and \( Y \), in [1,2] an algorithm is derived, to compute \( A^{-1} \) in the following form

\[
A^{-1} = A_0^{-1} - A_0^{-1} UR^{-1} V^T A_0^{-1},
\]

(3)

where \( R \in \mathbb{R}^{m \times m} \) is a diagonal matrix and \( U, V \in \mathbb{R}^{n \times m} \). The computational procedure is presented in Algorithm 1 below. We use Matlab-type notations and \( I_A, I_A^0 \) denote \( A^{-1} \) and \( A_0^{-1} \), correspondingly.

Algorithm 1 (Single vector ISM (SISM))

```matlab
for k = 1:m
    U(:,k) = X(:,k)
    V(:,k) = Y(:,k)
    for l = 1:k-1
        U(:,k) = U(:,k) - (V(:,l)'*I_A^0*X(:,k)) * R(l,l) * U(:,l)
        V(:,k) = V(:,k) - (Y(:,k)'*I_A^0*U(:,l)) * R(l,l) * V(:,l)
    end
    R(k,k) = 1/(1+V(:,k)'*I_A^0*X(:,k))
end
I_A = I_A^0 - I_A^0*U*R*V'*I_A^0
```

As we can see, SISM (Algorithm 1) consists of vector and matrix-vector (BLAS1) operations only, which are relatively less efficient than BLAS2 and BLAS3. A block implementation consisting of more efficient matrix operations can be expected to achieve better performance.

2.2 BLAS3 block version of the ISM algorithm

Because of the computational efficiency of BLAS3 operations, block versions of many numerical algorithms are capable of achieving higher performance than counterparts that rely on BLAS1
or BLAS2 operations. A block version of ISM has already been suggested in [2], similar to the method we now propose.

Consider X and Y to be of block-column form. Thus, let \( X = \{ X_k \}_{k=1,\ldots,p} \), \( Y = \{ Y_k \}_{k=1,\ldots,p} \) and \( X_k, Y_k \in \mathbb{R}^{n \times s_k} \) with \( \sum_{k=1}^{p} s_k = m \). Then, clearly, there holds that

\[
A = A_0 + \sum_{k=1}^{p} X_k Y_k^T.
\]  

(4)

Define \( A_k = A_{k-1} + X_k Y_k^T \) and assume the matrices \( R_k = I_{m_k} + Y_k A_{k-1}^{-1} X_k \) are nonsingular for \( k = 1, \ldots, p \). Applying formula (2), we obtain the following expression for the inverse of the matrices \( A_k \)

\[
A_k^{-1} = A_{k-1}^{-1} - A_{k-1}^{-1} X_k R_k^{-1} Y_k^T A_{k-1}^{-1}, \quad k = 1, \ldots, p.
\]  

(5)

Since \( A_p^{-1} = A^{-1} \), then applying formula (5) recursively we can have

\[
A^{-1} = A_0^{-1} - \sum_{k=1}^{p} A_{k-1}^{-1} X_k R_k^{-1} Y_k^T A_{k-1}^{-1}.
\]  

(6)

Then, another sequence of factors \( \{ U_k, V_k \}_{k=1,\ldots,p} \in \mathbb{R}^{n \times s_k} \)

\[
U_k = X_k - \sum_{i=1}^{k-1} U_i R_i^{-1} V_i^T A_0^{-1} X_k,
\]  

(7)

\[
V_k = Y_k - \sum_{i=1}^{k-1} V_i R_i^{-1} U_i^T A_0^{-1} Y_k,
\]  

(8)

are well defined. In addition, the relations

\[
A_{k-1}^{-1} X_k = A_0^{-1} U_k, \quad Y_k^T A_{k-1}^{-1} = V_k^T A_0^{-1},
\]

and

\[
R_k = I_{m_k} + Y_k^T A_0^{-1} U_k = I_{m_k} + V_k^T A_0^{-1} X_k.
\]

(9)

hold.

The above relations enable us to compute the factors \( U, V \) and \( R \) blockwise. Namely, for \( U = \{ U_1, U_2, \ldots, U_p \} \) and \( V = \{ V_1, V_2, \ldots, V_p \} \) with matrices \( U_k \) and \( V_k \) as columns, the inverse of \( A \) can be rewritten as

\[
A^{-1} = A_0^{-1} - A_0^{-1} U R^{-1} V^T A_0^{-1},
\]  

(10)

where \( R^{-1} = \text{diag}(R_1^{-1}, R_2^{-1}, \ldots, R_p^{-1}) \).

We see that in this case we have to invert matrix blocks \( R_k \) of some sizes \( s_k \). However, we can tune the block sizes in a suitable way so that the time to compute these inverses does not prevail the rest of the computations. Clearly, the block size parameter \( s_k \) may vary between the recursion steps. Without loss of generality, from now on we take \( s_k \) as constant. We present the pseudo-code of the above block algorithm in Algorithm 2.

**Algorithm 2 (BLAS3 Block ISM (BISM))**

\[ p = m/s; \ % p \text{ - number of blocks, } s \text{ - block size, } \text{Is} \text{ - identity of size } s \]
\[ U(:,1:s) = X(:,1:s); \ V(:,1:s) = Y(:,1:s) \]
\[ R0 = \text{Is} + Y(:,1:s)' \ast \text{IA0} \ast U(:,1:s); \ R(1:s,1:s) = \text{inv}(R0) \]

for \( k = 2:p \)
\[ X_{-}(k) = X(:, (k-1)*s+1:k*s) \]
\[ Y_{-}(k) = Y(:, (k-1)*s+1:k*s) \]
\[ W = \text{IA0} \ast X_{-}(k) \]
\[ P_{-}(k)(1:(k-1)*s,:) = V(:,1:(k-1)*s)' \ast W \]
\[ Q_{k}(1:(k-1)*s,:) = R(1:(k-1)*s,1:(k-1)*s) * P_{k} \]
\[ U_{k} = X_{k} - U(:,1:(k-1)*s) * Q_{k}(1:(k-1)*s,s) \]
\[ W = IA0' * Y_{k} \]
\[ P_{k}(1:(k-1)*s,:) = U(:,1:(k-1)*s)' * W \]
\[ Q_{k}(1:(k-1)*s,:) = R(1:(k-1)*s,1:(k-1)*s)' * P_{k} \]
\[ V_{k} = Y_{k} - V(:,1:(k-1)*s) * Q_{k}(1:(k-1)*s,s) \]
\[ R0 = Is + Y_{k}' * IA0*U_{k} \]
\[ R((k-1)*s+1:k*s,(k-1)*s+1:k*s) = inv(R0) \]
\[ U(:,(k-1)*s+1:k*s) = U_{k} \]
\[ V(:,(k-1)*s+1:k*s) = V_{k} \]

end

IA=IA0 - IA0*U*R*V'*IA0

The BISM algorithm requires four block-column matrices of size \( n \times s \), \( P_{k} \), \( Q_{k} \), \( U_{k} \), and \( V_{k} \), and one square matrix \( R0 \) of size \( s \times s \). For \( s = 1 \), BISM reduces to the single vector ISM algorithm. Since the BISM algorithm operates on blocks or submatrices of the original matrix, standard level-3 BLAS routines can be utilized, which are highly optimized and efficiently parallelized for modern high performance computers. Thus, the block ISM algorithm can be expected to be more efficient with respect to the single vector ISM especially for large matrices (see the numerical experiments in [3]). Here we choose to use the LU factorization to factorize the block \( R0 \) and then compute the columns of its inverse as the solutions of a linear system, with the identity matrix's columns as the right hand vectors. The computational complexity of BISM (Algorithm 2) is analyzed as follows.

- at the \( k \)th step \( k = 1,2,\cdots,p \)
  - computational work for \( P_{k} \): \( 2[2(k-1)ns^2] \).
  - computational work for \( Q_{k} \): \( 2[2(k-1)s^3] \).
  - computational work for \( U_{k}, V_{k} \): \( 2[2(k-1)ns^2 + ns] \).
  - total work by summing up \( k \)
    - computational work for \( P_{k} \): \( 2nm(m-s) \).
    - computational work for \( Q_{k} \): \( 2m(m-s)s \).
    - computational work for \( U_{k}, V_{k} \): \( 2nm(m-s) + 2nm \).
    - total work for \( R0^{-1} \): \( 5ms^2 + 2ns^2 \).
- Let \( m = \sigma n \), total computational complexity
  \[ 4\sigma^2n^3 + 2\sigma(1 + \sigma s - s)n^2 + 3\sigma s^2n. \]

Remark 1. As is seen from the assumptions, the ISM algorithm may break down - either when a zero scalar entry \( (R_{k,k}) \) is encountered in Algorithm 1 or a singular block \( (R0) \) is produced in Algorithm 2. We refer to [1] for a discussion on that issue and some techniques how to handle such a situation. In our numerical simulation, a breakdown of the ISM algorithm has not been encountered.

Remark 2. Even though \( A^{-1} \) is unique, the factors \( U, V \) and \( R \) in the resulting form of the exact inverse in (10) are not. These depend on the choice of \( A_{0} \), the order the columns of \( X \) and \( Y \) are used, the block factors \( s_k \) etc.

2.3 Block ISM with reduced memory footprint

BISM (Algorithm 2) stores the factors \( U \), and \( V \), each of them being a matrix of size \( n \times m \). This can be a problem when \( n \) is large and \( m \) is close to \( n \). We present a variation on the above algorithm that stores the product of \( U'R^{-1}V \) as a whole matrix \( H \) instead of separately storing the factor matrices, significantly reducing memory requirements when \( m \) is a large fraction of \( n \). For small \( m \), however, the storage of \( H \) requires more space than the individual factors. The trade-off for the decreased memory footprint is a higher computational complexity.
Algorithm 3 (Reduced Memory BLAS3 Block ISM (RMBISM))

\[ \begin{align*} 
    p &= \frac{m}{s}; \quad \% \text{p - number of blocks, s - block size, Is - identity of size s} \\
    U &= X(:,1:s); \quad V = Y(:,1:s); \\
    R0 &= \text{Is} + Y(:,1:s)'*I\!A_0*U; \quad \text{IR0} = \text{inv}(R0); \\
    H &= U*\text{IR0}*V'; \\
    \text{for } k=2:p, \\
        X_{\{k\}} &= X(:,\,(k-1)*s+1:k*s); \quad Y_{\{k\}} = Y(:,\,(k-1)*s+1:k*s); \\
        U &= X_{\{k\}} - H*I\!A_0*X_{\{k\}}; \quad V = Y_{\{k\}} - H'*I\!A_0'*Y_{\{k\}}; \\
        R0 &= \text{Is} + Y_{\{k\}}' * I\!A_0 * U; \quad \text{IR0} = \text{inv}(R0); \\
        H &= H + U*\text{IR0}*V'; \\
    \end{align*} \]

IA = I\!A_0 - I\!A_0*H*I\!A_0;

The total computational complexity of Algorithm 3 is found to be the following:

\[ \left(6 + \frac{1}{s}\right)\sigma n^3 + \left[4s(\sigma - 1) + 2\sigma - 1\right] n^2 + (5\sigma s - 2)sn, \]

where \( m = \sigma n \).

2.4 Tuning the block size \( s \)

The block size parameter \( s \) can be chosen arbitrarily in the range \([1, m]\), but the choice affects both performance and memory consumption. By varying \( s \), each of the two block ISM algorithms is affected differently. Here we present some theoretical reasoning regarding the choice of the block size, and in the next section we present experimental results for confirmation.

While the total computational complexity for the BISM (Algorithm 2) is minimized for a small \( s \), the algorithm is reduced to the single vector ISM for \( s = 1 \), forcing BLAS to use inefficient BLAS1 routines. BLAS libraries tend to work most efficiently with large matrices, i.e. for \( s \gg 1 \). Then, the optimal point of the tradeoff between computational complexity and BLAS library efficiency must be determined by numerical experiments and will depend on platform and implementation specifics.

The total computational complexity for the RMBISM (Algorithm 3) is minimized for \( s = m \) for all cases where \( \sigma \leq 0.5 \). This is the maximum possible size of \( s \), so in this case there is no performance tradeoff as above. The total computational complexity using the optimal block size is therefore \( (5\sigma^3 + 4\sigma^2 + 2\sigma)n^3 \). The existence of a linear term in \( \sigma \) dramatically decreases the effectiveness of this algorithm compared to BISM (Algorithm 2). For the case when \( \sigma = 1 \), the complexity of RMBISM is minimized when \( s = 1 \), leading to the same tradeoff as for the BISM algorithm. When \( 0.5 \leq \sigma \leq 1 \), the block size which minimizes computational complexity varies in a somewhat complicated way in the range \([1, m]\), which will be explored by numerical experiments.

As would be expected, both algorithms require more memory for larger block sizes. The memory requirements also strongly depend on \( m \). As the experiments below show, the memory requirements of the BISM algorithm scales more steeply than that of the RMBISM algorithm.

3 Implementing block ISM for multicore computers

Effective programs for multicore systems must support sufficient parallelism to fully exploit the available hardware. The BISM and RMBISM algorithms both feature very low level of parallelism on algorithm description level. To compute the block matrices \( U_k, V_k \) in Algorithm 2 or \( H \) in Algorithm 3, the previous ones must be computed first. However, the two algorithms consist almost exclusively of matrix products (DGEMM) and the computation of a matrix inverse (DGESV), operations which provide a high degree of parallelization. Both of these operations can be, and should be, performed in parallel through the use of parallel BLAS routines.

One goal with this work is to identify differences in the behavior of the BISM and RMBISM algorithms running on the GPU compared to the CPU in order to determine the utility of a
possible heterogeneous multicore version in the future. We write a straightforward implementation, by simply replacing DGEMM calls with calls to CUBLAS (Nvidia SDK 3.2) and compare its performance to the CPU codes.

4 Numerical experiments

In this section we use the ISM algorithms (Algorithms 1-3) to compute the exact inverse of an nonsingular matrix \( A \in \mathbb{R}^{n \times n} \), and we assume the matrix \( A \) is already written in the form \( A = A_0 + XY^T \). We choose \( A_0 = I_n \) and \( X, Y \in \mathbb{R}^{n \times m} \) created arbitrarily. The matrices \( X \) and \( Y \) are equally partitioned into two sets, i.e., \( X = [X_1, X_2, \ldots, X_p], \ Y = [Y_1, Y_2, \ldots, Y_p], \ {X_k,Y_k}_{k=1}^{p} \in \mathbb{R}^{n \times s}, \) where \( s = m/p \).

4.1 Speed optimisation

Based on the discussion in Section 2.4, we expect to see that the performance of Algorithms 2 and 3 vary with the block size \( s \). The following experiments are performed with Fortran implementations of the algorithms on a system with eight-core Intel Xeon X6550 processors. The BLAS routines are from the Sun Performance Library (see e.g., [5]).

We plot the runtime in Figures 1 and 2 (a) for some test problems and see that our expectations hold. For mid-to-low \( \sigma \), the impact of the block size on library efficiency is so significant that \( s = m \) is the optimal choice of block size regardless of algorithm and problem. For \( \sigma > 0.5 \), the optimal block size appears to be in the range \((0.02 \times m, 0.2 \times m)\).

We also see that the RMBISM algorithm, while improving on the single-vector ISM (SISM) at large \( s \), is less efficient than the BISM algorithm. We are convinced that RMBISM is appropriate for larger problems.

Another direct approach to compute \( A^{-1} \) could be to determine its columns as solutions of the linear systems \( AA^{-1} = I_n, \) where \( I_n \) is the identity matrix of size \( n \).

The parallel speedup of the two block algorithms is plotted in Figure 2 (b). We can see that both the two block algorithms achieve close to linear speedup, which means that they are benefiting fully from the parallelism inherent in BLAS3 operations.

4.2 Memory optimisation

As Figure 3 shows, the memory consumption of the BISM algorithm grows rapidly with \( m \), which motivated the development of the RMBISM algorithm, designed to consume less memory when \( m \) is relatively large. The memory footprint of RMBISM varies more slowly than that of BISM, yielding a memory savings of up to about 50\% when \( m = n \). With smaller \( m \), however, the usefulness of RMBISM is more limited (see Figure 4).

For a given \( n \) and \( m \), block size selection not only affects the performance but also the memory footprint. Our experimental results here can be illustrated by the following three cases:

- If \( m \sim n \), then memory may be a problem. If a small block size doesn’t sufficiently shrink the memory consumption of BISM, RMBISM will further reduce memory consumption.

- If \( m \sim n/2 \), then memory can still be a significant obstacle, but RMBISM is not very effective. Choosing a small block size can reduce the memory consumption of BISM by up to a factor of 3.

- If \( m < n/2 \), then memory consumption may not pose any problems, and RMBISM actually consumes more memory than BISM.

4.3 Numerical experiments using GPU

The GPU experiments are performed on a compute node consisting of two 8-core AMD Opteron 6220 (Bulldozer) processors at 3 GHz and a Nvidia Tesla M2050 GPU. Since the Bulldozer processors are configured such that two cores share a single FPU, using eight threads yields full hardware utilization and optimal performance for our CPU codes. These codes rely on the AMD Core Math
Fig. 1. Performance of BISM and RMBISM with varying blocksize.
Fig. 2. (a) Single-threaded runtimes of the algorithms with varying block size for a sample problem (logarithmic y scale). (b) Scaling behavior on a multicore system.
Fig. 3. Memory usage of both algorithms when varying $m$. Block size is chosen for optimal performance (see subsection 2.4 and subsection 4.1). Block size is equal to $m$ for $m \leq n/2$, and equal to $m/5$ when $m > n/2$.

Fig. 4. Up: Memory usage of algorithms for varying block sizes when $m = n$. Down: Memory usage of algorithms for varying block sizes when $m \sim n/2$ (double precision data).
Library (ACML) for their BLAS routines. The GPU codes use Nvidia CUBLAS, which lack the DGESV routines. We therefore perform only the DGEMM calls on the GPU, but these operations dominate the runtime to such an extent that the results remain meaningful (see Figure 5).

The Nvidia Tesla M2050 has a peak theoretical double precision performance of 515 GFLOPS, but CUBLAS performs at only about 200 GFLOPS [4], while the AMD processor has a peak theoretical double precision performance of almost 200 GFLOPS, which means that any speedup will be modest at best. As the results show in Figure 6, our GPU implementation achieves up to 20% faster than the CPU. We can also see that the performance of the GPU codes appears more sensitive to the block size choice than the CPU codes.

This is likely due to two factors: our straightforward implementation grossly ignores the cost of data transmission, and the profiler-reported device occupancy is only 33%. The first problem means that we perform unnecessary data movement between host memory and device memory, which can take up to 50% of the total runtime, according to profiling results. A proper implementation could decrease the data movement by up to a factor of 5. The issue with low occupancy may or may not actually impact performance depending on whether the global memory accesses on the device are successfully hidden, but this is worth considering when designing a better optimized implementation with respect to the GPU architecture.

![Figure 5](image_url). Proportion of runtime of the steps of the BISM algorithm. Block size is held constant, n varies.

5 Conclusions

In this paper we consider two parallelization strategies for the block ISM factorization. Our model problem is computing the exact inverse of a given dense nonsingular matrix. While this computation doesn’t always represent a real application for the ISM algorithm because of the potentially low accuracy of the resultant exact inverse, it is a useful problem for developing and analyzing new algorithms and implementations. Possible other applications are e.g. approximate inverses used for multiplicative preconditioners.

We present the parallel performance of the two block ISM factorizations, i.e., BISM and RMBISM, using a multicore CPU as well as a GPU. The main conclusions are that effective parallelization is quite easily implemented and the speedup is almost linear.

These results can be achieved over a wide range of choice in block size. Block size choice can be informed by memory considerations. A small block size can save up to 66% of the memory usage of BISM. For a larger problem, the RMBISM algorithm can save up to 50% of that of BISM.

The results from our straightforward GPU implementation, though modest, warrant a more serious implementation effort in the future.
Fig. 6. Speedup of running both algorithms on the Nvidia Tesla M2050 GPU compared to the Opteron 6620@3GHz running eight threads.

The effect of sparse matrices on the performance of the block ISM algorithm, data structures and parallelization techniques, as well as obtaining an approximate inverse of a dense or sparse matrix, needs to be further considered and is still in progress.

References