Efficient Model Order Reduction of Large-Scale Systems on Multi-core Platforms

P. Ezzatti[1], E. S. Quintana-Ortí[2], and A. Remón[2]

[1] Centro de Cálculo-Instituto de Computación, Universidad de la República (Montevideo, Uruguay)
`pezzatti@fing.edu.uy`

[2] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (Castellón, Spain)
`{quintana,remon}@icc.uji.es`

Abstract We propose an efficient implementation of the Balanced Truncation (BT) method for model order reduction when the state-space matrix is symmetric (positive definite). Most of the computational effort required by this method is due to the computation of matrix inverses. Two alternatives for the inversion of a symmetric positive definite matrix on multi-core platforms are studied and evaluated, the traditional approach based on the Cholesky factorization and the Gauss-Jordan elimination algorithm. Implementations of both methods have been developed and tested. Numerical experiments show the efficiency attained by the proposed implementations on the target architecture.

Key words: Model reduction, linear algebra, matrix inversion, SPD matrices, SVD-based methods, multi-core processors, BLAS.

1 Introduction

Model order reduction is a highly useful tool in the analysis and simulation of dynamical systems, control design, circuit simulation, structural dynamics, CFD, and many other disciplines involving complex physical models [1][2]. In particular, consider a linear time-invariant system corresponding, e.g., to a physical process, defined in state-space form by

$$\begin{array}{l} \dot{x}(t) = Ax(t) + Bu(t),\ t > 0,\ x(0) = x_0, \\ y(t) = Cx(t) + Du(t),\ t \geq 0, \end{array} \tag{1}$$

where $A \in \mathcal{R}^{n\times n}$, $B \in \mathcal{R}^{n\times m}$, $C \in \mathcal{R}^{p\times n}$, $D \in \mathcal{R}^{p\times m}$, $x_0 \in \mathcal{R}^n$ is the initial state of the system, and n is the order of the model. The goal for model reduction is to find a reduced-order realization

$$\begin{array}{l} \dot{x}_r(t) = A_r x_r(t) + B_r u(t),\ t > 0,\ x_r(0) = \hat{x}_0, \\ y_r(t) = C_r x_r(t) + D_r u(t),\ t \geq 0, \end{array} \tag{2}$$

where $A_r \in \mathcal{R}^{r\times r}$, $B_r \in \mathcal{R}^{r\times m}$, $C_r \in \mathcal{R}^{p\times r}$, $D_r \in \mathcal{R}^{p\times m}$, $\hat{x}_0 \in \mathcal{R}^n$ is the initial state of the system, r is the order of the new model, with $r \ll n$, and $\|y - y_r\|$

is "small". In other words, the purpose of model reduction is to obtain a new model with a smaller order (r), which can potentially replace the original model in subsequent computations yielding important time and resource cost savings. While just a few years ago, model reduction of dense large-scale models (state-space dimension n of $O(10^4 - 10^5)$) would have required the use of a cluster with a moderate number of nodes [3], modern multi-core processors provide enough computational power to tackle the major matrix computations appearing in model order reduction methods.

In previous works we have addressed the cases where the state-space matrix A is sparse [4], a general dense matrix [5], and a band matrix [6]. In this work we focus in the case when $-A$ is a dense symmetric positive definite (SPD) matrix. In this case, the structure and properties of the matrix can be exploited reporting important savings due to a significant reduction on the number of required operations.

The rest of the paper is structured as follows. Section 2 describes the state-of-the-art methods and libraries for model order reduction. Section 3 introduces the sign function method for the solution of Lyapunov equations. Then, in Section 4, we review the two approaches for computing the matrix inverse of an SPD matrix. Several high performance implementations of each method on multi-core processors are described and evaluated in Sections 5 and 6. Finally, a few concluding remarks and future work are offered in Section 7.

2 State-of-the-Art in methods and libraries

Model order reduction methods can be classified into two different families: moment matching-based methods and SVD-based methods (for a thorough analysis of these two families of methods, see [1]). The efficacy of model order reduction methods strongly relies on the problem and there is no technique that can be considered optimal in an overall sense. In general, moment matching methods employ numerically stable and efficient Arnoldi and Lanczos procedures in order to compute the reduced-order realizations. These methods, however, are specialized for certain problem classes and often do not preserve important properties of the system such as stability or passivity. On the other hand, SVD-based methods usually preserve these properties, and also provide bounds on the approximation error. However, SVD-based methods present a higher computational cost. In particular, all SVD-based methods require, as the most time consuming stage, the solution of two Lyapunov (or analogous matrix) equations. When balanced truncation is applied to (1), the Lyapunov equations that arise are

$$\begin{aligned} AW_c + W_cA^T + BB^T = 0, \\ A^TW_o + W_oA + C^TC = 0. \end{aligned} \tag{3}$$

In general, A is a stable matrix (i.e., all its eigenvalues have negative real part) and, therefore, matrices W_c, W_o are symmetric and positive semi-definite. Unfortunately, W_c, W_o are dense, square $n \times n$ matrices even if A is sparse. These equations can for instance be solved using direct Lyapunov solvers [7][8] from the SLICOT library[9], which allows the reduction of small LTI systems (roughly speaking, $n = 5,000$ on current desktop computers). Larger problems, with tens of thousands of state-space variables, can be reduced using the sign function-based methods in PLiCMR [10][3] on parallel computers [3]. The difficulties of exploiting the usual sparse structure of the matrices that appear at the Lyapunov equations using direct or sign function-based solvers limits the applicability of the SVD-based algorithms in these two libraries. However, those methods are completely based on high performance kernels from numerical linear algebra libraries, specifically BLAS and LAPACK.

3 The sign function method

The matrix sign function was introduced in [11] as an efficient tool to solve stable (standard) Lyapunov equations. The following variant of the Newton iteration for the matrix sign function [12] can be used for the solution of the Lyapunov equations (3):

Algorithm `CECLNC`:

```
$A_0 \leftarrow A$, $\tilde{S}_0 \leftarrow B^T$, $\tilde{R}_0 \leftarrow C$
$k \leftarrow 0$
repeat
    $A_{k+1} \leftarrow \frac{1}{\sqrt{2}}\left(A_k + A_k^{-1}\right)$
    $\tilde{S}_{k+1} \leftarrow \frac{1}{\sqrt{2}}\left[\tilde{S}_k,\ \ \tilde{S}_k(A_k^{-1})^T\right]$
    $\tilde{R}_{k+1} \leftarrow \frac{1}{\sqrt{2}}\left[\tilde{R}_k,\ \ \tilde{R}_k A_k^{-1}\right]$
    $k \leftarrow k + 1$
until convergence
```

On convergence, after j iterations, $\tilde{S} = \frac{1}{\sqrt{2}}\tilde{S}_j$ and $\tilde{R} = \frac{1}{\sqrt{2}}\tilde{R}_j$ of dimensions $\tilde{k}_o \times n$ and $\tilde{k}_c \times n$ are, respectively, full rank approximations of S and R, so that $W_c = S^T S \approx \tilde{S}^T \tilde{S}$ and $W_o = R^T R \approx \tilde{R}^T \tilde{R}$.

Two main reasons make the Newton iteration an appealing method for solving Lyapunov equations: its implementation is suitable to parallel programming and it usually presents a fast convergence rate, which is ultimately quadratic.

Each iteration of algorithm `CECLNC` requires $O(n^3)$ flops (floating-point arithmetic operations), where n is the dimension of matrix A. In particular, the following four operations are performed at each iteration:

1. Compute A_k^{-1}, the matrix inverse of an SPD matrix (n^3 flops)
2. Compute the addition of two symmetric matrices and scale the result (n^2 flops)
3. Compute $\tilde{S}_{k+1}$ via a matrix-matrix product ($n^2 \times \tilde{k}_o$ flops)
4. Compute $\tilde{R}_{k+1}$ via a matrix-matrix product ($n^2 \times \tilde{k}_c$ flops)

Therefore, most of the computational effort is concentrated on the calculation of the matrix inverse ${A_k}^{-1}$. This is reinforced from the numerical results reported in a previous work, where the same method is employed to solve a single Lyapunov equation (only steps 1 to 3 are required) with general dense coefficient matrices [5]. In that work, despite the use of a GPU to accelerate the computation of the inverse, this operation represented the 85% and 91% of the total computation time for two problems of dimension $5,177$ and $9,699$ respectively.

Also, there are high performance implementations for multi-core processors available for the rest of the operations involved in algorithm `CECLNC`, i.e. the matrix-matrix product (required at steps 3 and 4) and the matrix addition and scale (required at step 2) using e.g. the kernels provided in the BLAS library and OpenMP respectively.

This implies that, provided we can develop an efficient method for the computation of a matrix inverse, we can obtain a high performance Lyapunov solver and, thus, an efficient model-order reduction implementation. The rest of the paper is focused on the development of a high performance kernel for the inversion of an SPD matrix.

4 High performance matrix inversion of SPD matrices

In this section we survey two different algorithms for computing the inverse of an SPD matrix. The first algorithm is based on the computation of the Cholesky factorization, while the second algorithm employs the Gauss-Jordan elimination [13]. Both algorithms present the same computational cost, but the properties of the Gauss-Jordan elimination procedure are more suitable for parallel architectures.

4.1 Matrix inversion based on the Cholesky factorization

The traditional approach to compute the inverse of an SPD matrix $A \in \mathcal{R}^{n\times n}$ is based on the Cholesky factorization and consist of the three following steps:

1. Compute the Cholesky factorization $A = U^T U$, where $U \in \mathcal{R}^{n\times n}$ is an upper triangular matrix.
2. Invert the triangular factor $U \rightarrow U^{-1}$.
3. Obtain the inverse from the matrix-matrix product $U^{-1}U^{-T} = A^{-1}$.

By exploiting the symmetry of A, the computational and storage cost of the algorithm can be significantly reduced. In particular, as stated above, the computational cost is n^3 flops (compared, e.g., with the $2n^3$ flops required to invert a nonsymmetric matrix). In-place inversion of the matrix (i.e., inversion using only the storage provided in A) is possible which, besides, only references the upper triangular part of the matrix. However, for performance, A is stored as a full $n \times n$ matrix.

```
Algorithm: $[A] := \mathrm{GJE}_{\mathrm{BLK\_V1}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$ and $A_{BR}$ is $n \times n$
while $m(A_{TL}) < m(A)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
        where $A_{11}$ is $b \times b$

    $W := -A_{00} \cdot A_{01}$                                            SYMM
    $A_{11} := A_{11} + A_{01}^T \cdot A_{01}$                             GEMM
    $A_{11} := \mathrm{CHOL}(A_{11})$                                      POTRF
    $\mathrm{TRIU}(A_{11}) := \mathrm{TRIU}(A_{11})^{-1}$                  TRTRI
    $W := W \cdot A_{11}$                                                  TRMM
    $A_{01} := W \cdot A_{11}^T$                                           TRMM
    $A_{00} := A_{00} + W \cdot W^T$                                       SYRK
    $A_{11} := \mathrm{TRIU}(A_{11}) \cdot \mathrm{TRIU}(A_{11})^T$        LAUUM

    Continue with
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
endwhile
```

Figure 1. Blocked algorithm for matrix inversion of SPD matrices via GJE (Variant 1).

4.2 Matrix inversion based on the Gauss-Jordan elimination algorithm

The Gauss-Jordan elimination (GJE) algorithm is, in essence, a reordering of the computations performed by the traditional approach. Thus, it presents the same computational cost. The reordering reduces notably the number of sweeps through the matrix (and, therefore, the number of memory accesses) as well as yields a much more balanced workload distribution in a parallel execution [14].

This method can also be carefully designed to exploit the symmetric structure of the matrix and produce in-place results.

Figures 1 and 2 show two blocked GJE-based algorithms using the FLAME notation [15][16]. There, $m(\cdot)$ stands for the number of rows of its argument; $\mathrm{TRIU}(\cdot)$ returns the upper triangular part of a matrix; and "$\star$" specifies blocks in the lower triangular part of the matrix, which are not referenced. We believe the rest of the notation is intuitive. Next to each operation, we provide the name

```
Algorithm: $[A] := \text{GJE}_{\text{BLK\_V2}}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$ and $A_{BR}$ is $n \times n$
while $m(A_{TL}) < m(A)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
        where $A_{11}$ is $b \times b$

    $A_{11} := \text{CHOL}(A_{11})$                          POTRF
    $\text{TRIU}(A_{11}) := \text{TRIU}(A_{11}^{-1})$        TRTRI
    $A_{01} := A_{01} \cdot A_{11}$                          TRMM
    $A_{00} := A_{00} + A_{01} \cdot A_{01}^T$               SYRK
    $A_{01} := A_{01} \cdot A_{11}$                          TRMM
    $A_{12} := A_{11}^{-T} \cdot A_{12}$                     TRMM
    $A_{22} := A_{22} - A_{12}^T \cdot A_{12}$               SYRK
    $A_{02} := A_{02} - A_{01} \cdot A_{12}$                 GEMM
    $A_{12} := -(A_{11} \cdot A_{12})$                       TRMM
    $A_{11} := A_{11} \cdot A_{12}^T$                        LAUUM

    Continue with
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
endwhile
```

Figure 2. Blocked algorithm for matrix inversion of SPD matrices via GJE (Variant 2).

of the BLAS kernel that is employed to perform the corresponding operation. In both algorithms the inverse overwrites the initial matrix.

Up to eight operations are carried out at each iteration of the algorithm in Figure 1. Two factors will limit the performance of its parallel implementation. First, data dependencies serialize the execution of most of the operations. Second, except the update of block A_{00}, the computations involve uniquely blocks of reduced size (taking into account that, for performance reasons, the value of the block size b is chosen to be small compared with n). This limits the inherent parallelism of the variant, specially during the first iterations of the loop, when A_{00} is also a small block.

Figure 2 shows a second variant of the GJE algorithm where all the elements of the upper part of the matrix are updated at each iteration. This results in a constant computational effort during the iterations. Again, data dependencies serialize the execution of most operations. Thus, parallelism can only be extracted from within the invocation of single operations. In this variant, the updates of blocks A_{00} and A_{22} concentrate most of the computations, while the rest of operations involve small blocks. This implementation presents two advantages respect the previous variant:

– It does not require any additional work space.
– The computational cost of each iteration is constant.

5 High performance implementations

5.1 Implementations based on the Cholesky factorization

The algorithm based on the Cholesky factorization for the computation of the inverse of an SPD matrix (see Section 4.1) is composed of three steps that must be executed in order. This means that parallelism can only be extracted inside the execution of each step.

The Intel MKL library [17] offers kernels for the Cholesky factorization of an SPD matrix (routine `potrf`, Step 1) and the inversion from its triangular factors (routine `potri`, Steps 2 and 3). The use of a multi-thread version of MKL offers parallelism and efficiency for the execution of both routines on a multi-core CPU.

5.2 Implementations based on the Gauss-Jordan elimination

In this subsection we describe the two variants of the GJE algorithm introduced in Section 4.

In both variants, most of the computations are cast in terms of matrix-matrix products. In particular, the operation that involves a higher number of flops is a symmetric rank-k update (a special case of the matrix-matrix product). The MKL library offers high performance implementations of this computational kernel as well as the remaining operations present in algorithms GJE$_{BLK\_V1}$ and GJE$_{BLK\_V2}$. Routines GJE_v1 and GJE_v2 implement those algorithms using MKL kernels. Parallelism is obtained, once more, within the execution of each single operation invoking the multi-threaded version of MKL.

6 Experimental results

In this section we evaluate the performance and scalability of the implementations presented in Section 5.

All experiments in this section were performed using IEEE single precision arithmetic. Results are shown for SPD matrices of dimension 1,000, . . . , 15,000. Different algorithmic block-sizes were tested (1024, 512, 256, 128, 64 and 32) but, for simplicity, we only report the performance obtained with the optimal algorithmic block-size.

The platform employed at the experiments consists of four Intel Xeon X7550 processors (with 8 cores per processor) running at 2.0 GHz. More details from the hardware can be found in Table 1. Kernels from the Intel MKL 11.0 multi-threaded implementation of BLAS and LAPACK are used for most of the computations.

Processors	#cores	Freq. (GHz)	L2 (MB)	Memory (GB)
Intel Xeon X7550	(8×4) 32	2.0	18	124

Table 1. Hardware employed in the experiments.

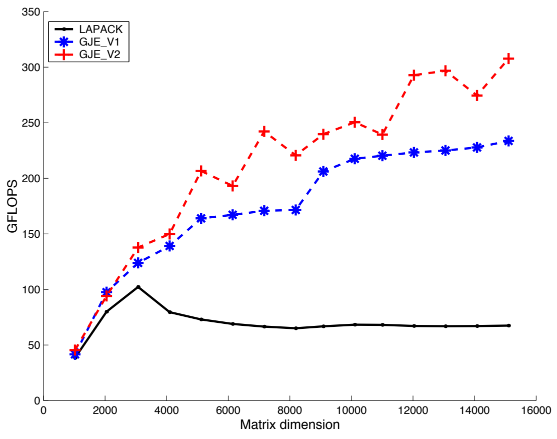

Figure 3. Performance of the matrix inversion codes.

Figure 3 shows the performance attained by the LAPACK and the GJE based implementations described in Section 5 using 32 threads (one thread per core on the target platform). The implementation of the first variant (GJE_V1) is notoriously more efficient than LAPACK, specially for large matrices (e.g., it is approximately 8× faster for matrices of dimension 15,000). However, the best performance is obtained by the implementation of the second variant of the algorithm (GJE_V2). It achieves more than 300 GFLOPS for matrices of dimension 15,000, being more than 10× faster than LAPACK. To sum up, both GJE implementations offer better performance than LAPACK but, due to the properties of the second variant, its implementation renders a higher efficiency.

Figure 4 shows the results obtained by the GJE_V1 implementation using 1, 2, 4, 8, 16 and 32 threads. The use of more threads increments the performance considerably, except for the inversion of matrices of dimension up to 8,000 using more than 16 threads. These results demonstrate the scalability of GJE_V1.

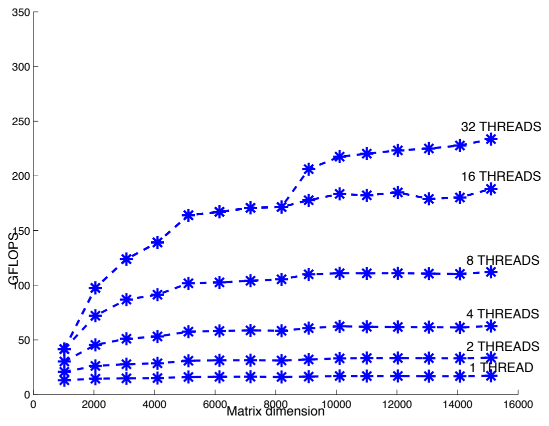

Figure 4. Performance of the matrix inversion using the GJE_V1 version.

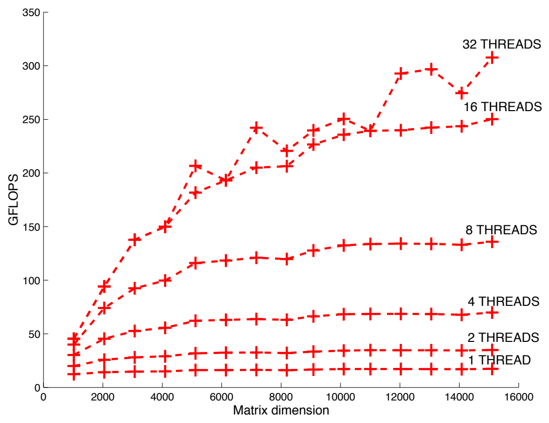

Figure 5. Performance of the matrix inversion using the GJE_V2 version.

Finally, Figure 5 is the analogous for the GJE_V2 variant. Again the results obtained demonstrate the scalability of the developed implementation.

7 Concluding remarks

We have studied the inversion of symmetric positive definite matrices. This operation appears in model reduction and requires a high computational effort. This asks for the use of high performance architectures like multi-core CPUs. The study includes the evaluation of two different algorithms, the traditional algorithm based on the Cholesky factorization and the GJE algorithm, more suitable for parallel architectures.

Several implementations are presented for each algorithm and the studied architecture, which extract parallelism from computational kernels in multi-threaded implementations of BLAS, like Intel MKL.

Experimental results demonstrate that higher performance is attained by the routines based on the GJE algorithm. This algorithm exhibits a remarkable scalability in all its implementations.

Acknowledgments

The authors would like to thank Francisco Igual for his technical support, and Manuel Ujaldon from the University of Malaga (Spain) for granting the access to the hardware facilities employed in the experimental evaluation. Enrique S. Quintana-Ortí and Alfredo Remón were supported by the CICYT project TIN2008-06570-C04.

References

1. A. Antoulas, *Approximation of Large-Scale Dynamical Systems.* SIAM Publications, 2005.
2. V. M. P. Benner, *Dimension Reduction of Large-Scale Systems.* Springer-Verlag Berlin Heidelberg, 2005.
3. P. Benner, E. S. Quintana-Ortí, and G. Quintana-Ortí, "State-space truncation methods for parallel model reduction of large-scale systems," *Parallel Computing*, vol. 29, no. 11-12, pp. 1701 – 1722, 2003.
4. J. M. Badía, P. Benner, R. Mayo, and E. S. Quintana-Ortí, "Parallel algorithms for balanced truncation model reduction of sparse systems," in *Applied Parallel Computing*, ser. Lecture Notes in Computer Science.
5. P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana-Ortí, and A. Remón, "A mixed-precision algorithm for the solution of Ļyapunov equations on hybrid CPU-̧GPU platforms," *Parallel Computing*, 2010.
6. A. Remón, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Solution of band linear systems in model reduction for VSLI circuits," in *Scientific Computing in Electrical Engineering*, ser. Mathematics in Industry.
7. R. H. Bartels and G. W. Stewart, "Solution of the matrix equation ax + xb = c [f4]," *Commun. ACM*, vol. 15, pp. 820–826, September 1972.

8. S. J. Hammarling, "Numerical Solution of the Stable, Non-negative Definite Lyapunov Equation," *IMA Journal of Numerical Analysis*, vol. 2, no. 3, pp. 303–323, 1982.
9. SLICOT (Control and Systems Library).
10. P. Benner, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Balanced Truncation Model Reduction of Large-Scale Dense Systems on Parallel Computers," *Mathematical and Computer Modelling of Dynamical Systems*, vol. 6, pp. 383–405, 2000.
11. J. Roberts, "Linear model reduction and solution of the algebraic Riccati equation by use of the sign function," vol. 32, pp. 677–687, 1980, (Reprint of Technical Report No. TR-13, CUED/B-Control, Cambridge University, Engineering Department, 1971).
12. P. Benner, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Solving linear-quadratic optimal control problems on parallel computers," *Optimization Methods Software*, vol. 23, pp. 879–909, December 2008.
13. G. Golub and C. Van Loan, *Matrix Computations*, 3rd ed. Baltimore: Johns Hopkins University Press, 1996.
14. P. Bientinesi, B. Gunter, and R. A. van de Geijn, "Families of algorithms related to the inversion of a symmetric positive definite matrix," *ACM Trans. Math. Softw.*, vol. 35, pp. 3:1–3:22, July 2008.
15. P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn, "The science of deriving dense linear algebra algorithms," *ACM Trans. Math. Softw.*, vol. 31, pp. 1–26, March 2005.
16. J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "Flame: Formal linear algebra methods environment," *ACM Transactions on Mathematical Software*, vol. 27, no. 4, pp. 422–455, Dec. 2001.
17. Intel Corporation., `http://www.intel.com/`.