Computers and Mathematics with Applications 71 (2016) 693–711

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

Sparse approximate inverse preconditioners on high performance GPU platforms☆

Daniele Bertaccini [a,*], Salvatore Filippone [b]

[a] Department of Mathematics, University of Rome "Tor Vergata", via della Ricerca Scientifica 1, I-00133, Roma, Italy
[b] School of Aerospace, Transport and Manufacturing, Bldg 52, Cranfield University, Cranfield, MK43 0AL, United Kingdom

ARTICLE INFO

Article history:
Received 20 March 2015
Received in revised form 9 October 2015
Accepted 12 December 2015
Available online 28 January 2016

Keywords:
Preconditioners
Approximate inverses
Sparse matrices
GPU

ABSTRACT

Simulation with models based on partial differential equations often requires the solution of (sequences of) large and sparse algebraic linear systems. In multidimensional domains, *preconditioned* Krylov iterative solvers are often appropriate for these duties. Therefore, the search for efficient preconditioners for Krylov subspace methods is a crucial theme. Recent developments, especially in computing hardware, have renewed the interest in *approximate inverse preconditioners* in factorized form, because their application during the solution process can be more efficient. We present here some experiences focused on the approximate inverse preconditioners proposed by Benzi and Tůma from 1996 and the sparsification and inversion proposed by van Duin in 1999. Computational costs, reorderings and implementation issues are considered both on conventional and innovative computing architectures like Graphics Programming Units (GPUs).

1. Introduction

The numerical approximation of the vast majority of mathematical models from applied science and engineering requires solving linear algebraic systems that become every day larger and larger. In particular, this happens for finite volumes, finite elements or finite differences schemes solving models based on partial differential equations. Very often, most of the computing time is spent in the solution of those linear systems by direct or iterative algorithms. However, direct methods sometimes can be less appropriate than iterative for multidimensional problems and Krylov iterative methods can give a viable approach especially on parallel architectures. Preconditioning is often essential because without it the convergence of iterative solvers can be too slow for practical purposes.

Preconditioning a linear system $Ax = b$ by a direct approximation of A^{-1}, i.e. by using approximate inverse preconditioners, has been quite popular in the last two decades; see, e.g. [1] and references therein.

From an implementation point of view, preconditioning by an approximate inverse is carried out by sparse matrix by vector multiplications, the same numerical kernel that forms the core of any Krylov subspace method. On highly parallel computing architectures such as the GPUs it is quite possible to implement efficiently matrix–vector multiplication, but GPU implementations for triangular systems are much less efficient [2–4].

Aside from the potential to exploit parallel hardware potentialities, there are also frameworks where sparse approximate inverse is required. An example is the updates for inverse incomplete factorization preconditioners for solving cheaply

☆ Work supported in part by grants MIUR-PRIN no. 20083KLJEZ, Italian Ministry of Health no. RF-2009-1470310 and by an INDAM-GNCS grant 2012. The CINECA Supercomputer Facilities (Standard HPC Grant 2012 with projects "Exa-PSBLAS" and "GPU-PSBLAS") are also gratefully acknowledged.
* Corresponding author.
E-mail addresses: bertaccini@mat.uniroma2.it (D. Bertaccini), salvatore.filippone@cranfield.ac.uk (S. Filippone).

http://dx.doi.org/10.1016/j.camwa.2015.12.008
0898-1221/

sequence of large and sparse linear systems proposed in [5–8]. We recall that sequences of large and sparse linear systems arise, e.g., in the numerical solution of nonlinear algebraic systems, ordinary and partial differential equations, and nonlinear optimization.

Among the various approaches for computing sparse approximate inverses that have been proposed in the literature through the years we can find:

- Minimization of the residual norm;
- Approximation by a matrix polynomial;
- Inexact inversion of sparse triangular factors;
- Incomplete biconjugation.

This paper will deal with the last two approaches. See [9] for a recent GPU implementation of the first one.

The remainder of the paper is organized as follows. In Section 2 we recall briefly the sparse approximate inverses based on inversion and sparsification of incomplete factorizations proposed in [10] and in Section 3 the approximate inverse preconditioners based on incomplete biconjugation proposed by Benzi and Tůma. In Section 4 we give a brief discussion of the main features of the algorithms for the underlying approximate inverse factorization to prepare for some experimental results; Section 5 holds a description of a number of software and hardware issues. Finally, in Sections 6 and 7 we report some numerical tests and conclusions and perspectives.

2. Sparse inversion of sparse factors

A strategy for computing a sparse approximate inverse based on incomplete factorizations is advocated by A. van Duin [10]. The idea is to start from a sparse (approximate) factorization

$$A \approx LDU,$$

where L and U are unit lower and upper triangular matrices and D is diagonal. Let us now describe the process with respect to the upper factor U; the lower factor L can be treated in a completely analogous way.

The triangular matrix U can be represented as

$$U = I + \sum_{i=1}^{n-1} e_i u_i^T, \tag{1}$$

where u_i^T is the ith row of the strictly upper part of U, i.e.

$$u_i(j) = 0 \quad \forall j \le i.$$

Given this structure, it is also possible to express the same matrix in product form as

$$U = \prod_{i=n-1}^{1} (I + e_i u_i^T), \tag{2}$$

where each term in the product is an elementary transformation. Since each term is easily invertible, we obtain the following expression

$$U^{-1} = \prod_{i=1}^{n-1} (I - e_i u_i^T), \tag{3}$$

but given that this matrix is also upper triangular we can express it in the form

$$U^{-1} = I + \sum_{i=1}^{n-1} e_i \hat{u}_i^T. \tag{4}$$

Combining (4) and (3) we derive an expression for $\hat{u}_i^T$:

$$\hat{u}_i^T = -u_i^T \prod_{j=i+1}^{n-1} (I - e_j u_j^T). \tag{5}$$

We can recast this expression in the Algorithm 1 for computing U^{-1}. To turn this into an effective preconditioning strategy we need however to apply some form of "drop strategy" to preserve the sparsity of the resulting factors. Let us emphasize that, as noted by [11,12], the usage of a factored form to approximate A^{-1} can be helpful in this respect: an irreducible sparse matrix has an inverse that is typically full, but, under suitable conditions, its inverse can be approximated reasonably by the product of two sparse matrices. Therefore, we can expect to reap some benefits in this respect.

2.1. Positional drop strategies

The most common positional drop strategy is based on the concept of level of fill; the resulting algorithm bears many similarities with the incomplete factorizations based on levels of fill, and is shown in Algorithm 2. Throughout the paper we

Algorithm 1 A General Sparse Triangular Inverse

1: **for** $j = 1$ to $n - 1$ **do**
2: $\hat{u}_i^T \leftarrow -u_i^T$
3: $j \leftarrow$ location of first nonzero in $\hat{u}_i^T$
4: **while** $j < n$ **do**
5: $\alpha \leftarrow -\hat{u}_i^T e_j$
6: $\hat{u}_i^T \leftarrow \hat{u}_i^T + \alpha u_j^T$
7: $j \leftarrow$ location of next nonzero in $\hat{u}_i^T$
8: **end while**
9: **end for**

Algorithm 2 Positional Fill Level Triangular Inverse, or *INVK*

1: $level_{ij} \leftarrow 0$ if $a_{ij} \neq 0$, ∞ otherwise;
2: **for** $j = 1$ to $n - 1$ **do**
3: $\hat{u}_i^T \leftarrow -u_i^T$
4: $j \leftarrow$ location of first nonzero in $\hat{u}_i^T$
5: **while** $j < n$ **do**
6: **if** $level_{ij} \leq p$ **then**
7: $\alpha \leftarrow -\hat{u}_i^T e_j$
8: $\hat{u}_i^T \leftarrow \hat{u}_i^T + \alpha u_j^T$
9: update fill levels $level_{ik} = \min(level_{ik}, level_{ij} + 1)$
10: **else**
11: $\hat{u}_i^T(j) \leftarrow 0$
12: **end if**
13: $j \leftarrow$ location of next nonzero in $\hat{u}_i^T$
14: **end while**
15: **end for**

will refer to this method as *INVK*. In the overall process, the fill level will have to be specified for both the *LDU* incomplete factorization as well as for the sparse triangular inversion. In the sequel we will split the specification in two steps, so that an *INVK*(I, J) is to be interpreted as allowing a fill level of I in the factorization phase, and then an additional fill level of J in the inversion phase.

2.2. Numerical drop strategies

In a manner completely analogous to the previous subsection, the numerical drop strategy for the approximate inverse shown in Algorithm 4 (shown in Section 2.3) can be implemented with algorithmic steps very similar to the factorization phase; the resulting algorithm will be called here *INVT*. In an actual implementation, it is normal practice to use the threshold ϵ as a relative value with respect to the norm of the relevant row; this reduces the dependency on the scaling of the original problem.

2.3. Factorization and inversion implementation

Let us review the basic incomplete factorization algorithm with threshold (Algorithm 3), described in [13]: Each iteration of the main factorization loop may be partitioned in three phases:

1. A *copy-in* phase at step 2 where we take the ith row of matrix A and expand it in a full row w;
2. A factorization loop 3 where we apply all the needed updates from the previous phases of the factorization and the first dropping rule 5;
3. A *copy-out* phase at steps 10–13 in which we also apply the second dropping rule.

The two dropping rules are slightly different in nature. The dropping rule at step 5 is based on the comparison of w_k with a user specified threshold. The second dropping rule at step 10 is more complicated: first, we perform a comparison with a threshold; then, we keep only the p elements of largest absolute value among those which were not dropped.

As written this algorithm is hopelessly expensive: the nested loop 3 on k is executed $i - 1$ times and for each iteration we execute a vector update of size $n - i$, thus giving an overall cost that is quadratic in n. To keep the cost under control the first thing to do is to run the loop 3 on just the values that correspond to non zero entries in the current row w. The set of nonzeros in w is easily identified when it is initialized at step 2, but the problem is that it is altered during the course of the loop itself. An efficient loop would be described as follows:

Algorithm 3 Incomplete factorization with threshold

1: **for** $i = 1, \ldots, n$ **do**
2: $w \leftarrow a_{i*}$
3: **for** $k = 1, \ldots, i-1$ and $w_k \neq 0$ **do**
4: $w_k \leftarrow w_k / d_k$
5: Apply a drop rule to w_k
6: **if** $w_k \neq 0$ **then**
7: $w \leftarrow w - w_k u_{k*}$
8: **end if**
9: **end for**
10: Apply a drop rule to row w
11: $l_{ij} \leftarrow w_j \quad j = 1, \ldots, i-1$
12: $l_{ii} \leftarrow 1$; $d_i \leftarrow w_i$; $u_{ii} \leftarrow 1$;
13: $u_{ij} \leftarrow w_j \quad j = i+1, \ldots, n$
14: $w \leftarrow 0$
15: **end for**

Algorithm 4 Numerical Fill Drop Triangular Inverse or *INVT*

1: **for** $j = 1$ to $n-1$ **do**
2: $\hat{u}_i^T \leftarrow -u_i^T$
3: j location of first nonzero in $\hat{u}_i^T$
4: **while** $j < n$ **do**
5: $\alpha \leftarrow -\hat{u}_i^T e_j = -\hat{u}_i^T(j)$
6: **if** $|\alpha| > \epsilon$ **then**
7: $\hat{u}_i^T \leftarrow \hat{u}_i^T + \alpha u_j^T$
8: **else**
9: $\hat{u}_i^T(j) \leftarrow 0$
10: **end if**
11: j location of next nonzero in $\hat{u}_i^T$
12: **end while**
13: Drop elements in $\hat{u}_i$ as necessary to achieve the desired number of nonzeros.
14: **end for**

At each iteration, select the next lowest index k among the nonzero entries of w; update w_k and if it is not dropped then:

- use it to update the rest of w with u_{k*}, possibly adding new nonzeros to w and thus new indices to the candidate set in w;
- Add w_k to the set of entries in w to be considered for the second drop rule and copying out in steps 10–13.

A subtle and interesting problem surfaces with the seemingly innocuous statement at step 14: if we naively zero out the entire row w, we are accessing all its entries, and therefore incur an $O(n)$ cost. This $O(n)$ cost is incurred at each iteration of the outer loop, which runs through all n columns: thus, we would be reintroducing an overall quadratic cost for our algorithm. To overcome this issue we make use of the drop rule and copy-out steps 10–13: here, we can zero coefficients selectively, as we run through the entries of w that were not annihilated at step 5. Then, it only remains to initialize to zero the data area for w before entering the main loop; this is an example of how small details can sometimes imperil the implementation of an otherwise reasonable algorithm.

Let us now turn to the issue of implementing the sparse inversion of a sparse triangular factor. By comparing the threshold based version presented in Algorithm 4 with the factorization discussed above, it is clear that it is possible to implement it with the same "elementary" operators, and specifically:

- During the copy-in phase in step 2 we initialize the set of nonzero entries for the current row $\hat{u}_i$;
- During the update phase in step 7 we also insert the relevant indices into the set to ensure that the retrieval of the next nonzero at step 11 is performed efficiently;
- At the end of the inner loop, we perform a copy-out operation bringing the row $\hat{u}_i$ into its desired final state, copying the largest entries up to the maximum allowed number of nonzeros.

From the above discussion it is clear that for both the factorization and the inversion phases we are looking for ways to implement efficiently the following two operations on a set with an order relation:

1. Select and remove the lowest ranked element from a set;
2. Add an element to the set.

One possible and efficient solution relies on the *Partially Ordered Set Abstract Data Type* [14]. This guarantees that both insertion of a new element and deletion of the lowest ranked element can be performed with a cost $O(\log(|S|))$ where $|S|$ is the cardinality of the set S.

The copy-out operations in both factorization and inversion can be again implemented by making use of a partially ordered set, but this time we are interested in a set that is ordered based on the absolute value of its entries, since the second dropping rule in both phases states that we are to keep the p largest entries of the current row.

With the appropriate implementation for this data structure we are now in a position to estimate the cost of building an approximate inverse as in Algorithm 4.

Theorem 1. *Let* nz_u *be the average number of nonzeros per row in* u *and let* $\mathrm{nz}_{\hat{u}}$ *be defined similarly; assume moreover the bounds*

$$|S| \leq \gamma \mathrm{nz}_u, \tag{6}$$

$$\mathrm{nz}_{\hat{u}} \leq \beta \mathrm{nz}_u, \tag{7}$$

where $|S|$ *is the maximum size of the set of entries in any of the* $\hat{u}_i$ *before the application of the drop rule* 13*. Then the cost of Algorithm* 4 *is*

$$O(\gamma \beta n \cdot \mathrm{nz}_u^2(1 + \log(\gamma \mathrm{nz}_u))). \tag{8}$$

Proof. Given the bound (6), the term $\gamma n \cdot \mathrm{nz}_u$ follows easily from the nesting of the two outer loops. For the remaining factor $\beta \mathrm{nz}_u(1 + \log(\gamma \mathrm{nz}_u))$, consider statement 7: this is executed whenever the nonzero in $\hat{u}_i$ is above threshold, and we would expect this to happen a number of times within a (moderate) factor times the size of u_i. On each execution, statement 7 requires nz_u floating-point operations to execute the sparse AXPY, plus $\mathrm{nz}_u \log(|S|)$ operations to update the set S with the (possibly new) nonzero entries. Using again bound (6) gives the desired result.

The result relies on the two crucial assumptions about the size of the sets involved, and specifically that both β and γ are small constants. Assumption (7) is easier to justify: it simply expresses the fact that in many applications we would normally like to have a preconditioner that has a number of nonzeros of the same order as the coefficient matrix A, hence we would have $\beta \approx 1$. Since the number of nonzeros can be enforced at step 13, this is not a problem.

Assumption (6) is a bit more complex: it relies on the interaction between the profile of u and $\hat{u}$. In particular, the hypothesis (6) is considered plausible, at least for important classes of problems, see Theorem 4 in [12]. The latter gives an exponential decaying argument for the entries of the inverse of the Cholesky factor. The application of the dropping rules at 6 and 13 in Algorithm 4 is also acting to keep the number of elements in the set S under control.

The cost of the sparse inversion of sparse factors INVK and INVT has also been analyzed in the paper where they were proposed [10]. The approximate inversion for the upper factor is estimated at

$$C_{\mathrm{invrt}} = O\left(\mathrm{nz}_{\hat{U}} \frac{\mathrm{nz}_U}{n}\right)$$

where nz_U is the number of nonzeros above the main diagonal in U and likewise for the other quantities.

Note that the upper bound for the first term $\mathrm{nz}_{\hat{U}}$ is given by the product $n\beta \mathrm{nz}_u$ while the second term is nz_u. This estimate is then equivalent to our estimate (8) under the mild assumption that $\log(\gamma \mathrm{nz}_u)$ is bounded by a small constant.

3. AINV: a method based on incomplete biconjugation

The method we are about to discuss was proposed in [15] and later extended in [16]. It is based on the observation that if a matrix $A \in \mathbb{R}^{n \times n}$ is nonsingular, and if we have two vector sequences $\{z_i, i = 1 \dots n\}$ and $\{w_i, i = 1 \dots n\}$ which are A-biconjugate, i.e. $w_i^T A z_j = 0$ if and only if $i \neq j$, we can express the biconjugation relation as follows:

$$W^T A Z = D = \begin{pmatrix} p_1 & 0 & \dots & 0 \\ 0 & p_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & p_n \end{pmatrix} \tag{9}$$

where $p_i = w_i^T A z_i \neq 0$. Thus, W and Z must be nonsingular, since D is nonsingular. Therefore, we have

$$A = W^{-T} D Z^{-1}$$

from which it readily follows that

$$A^{-1} = Z D^{-1} W^T. \tag{10}$$

If W and Z are triangular, then they are actually the inverses of the triangular factors in the familiar LDU decomposition, as can be easily seen by comparing the two expressions

$$A = LDU$$

and

$$A = W^{-T} D Z^{-1}.$$

Algorithm 5 Biconjugation

1: $w_i^{(0)} \leftarrow z_i^{(0)} \leftarrow e_i \qquad 1 \le i \le n$
2: **for** $i = 1, \ldots, n$ **do**
3: **for** $j = i, i+1, \ldots, n$ **do**
4: $p_j^{(i-1)} \leftarrow a_i^T z_j^{(i-1)}; \qquad q_j^{(i-1)} \leftarrow c_i^T w_j^{(i-1)}$
5: **end for**
6: **for** $j = i+1, \ldots, n$ **do**
7: $z_j^{(i)} \leftarrow z_j^{(i-1)} - \left(\frac{p_j^{(i-1)}}{p_i^{(i-1)}}\right) z_i^{(i-1)}; \qquad w_j^{(i)} \leftarrow w_j^{(i-1)} - \left(\frac{q_j^{(i-1)}}{q_i^{(i-1)}}\right) w_i^{(i-1)}$
8: **end for**
9: **end for**
10: $w_i \leftarrow w_i^{(i-1)},\ z_i \leftarrow z_i^{(i-1)},\ p_i \leftarrow p_i^{(i-1)},\ 1 \le i \le n$

There are infinitely many biconjugate sequences $\{w\}$ and $\{z\}$: to find any one of them it is sufficient to apply a biconjugation procedure to the appropriate pair of nonsingular matrices $W^{(0)}, Z^{(0)} \in \mathbb{R}^{n\times n}$. From a computational point of view one can start with $W^{(0)} = Z^{(0)} = I$, thus obtaining the procedure in Algorithm 5, where a_i^T is the ith row of A and c_i^T is the ith row of A^T. If the procedure reaches completion without breakdowns, i.e. all the diagonal elements are nonzero, then the resulting matrices W and Z will be triangular, again giving the explicit inverses of L and U. Thus, we can conclude that for symmetric positive definite matrices the process will not break down. Another interesting feature of Algorithm 5 is that the process for building W can proceed independently of the process to build Z.

To turn Algorithm 5 into a practical procedure, we need to "sparsify" the resulting W and Z by dropping elements in the vectors w_i and z_i. In principle this could be done at the end of Algorithm 5, but this would mean storing the matrices W and Z in full until the very end. Thus in practice the sparsification has to happen at all updates to the vectors w and z.

Similarly to the case of the incomplete factorization, it is possible to prove [17] that the incomplete inverse factorization exists (in exact arithmetic) when A is an H-matrix.

An important point to be noted is that, despite the many similarities, there is a noticeable difference with the case of incomplete factorizations. It is well known that if A is an M-matrix, then the incomplete factorization induces a regular splitting $A = \hat{L}\hat{U} - R$, i.e. $\rho(I - \hat{U}^{-1}\hat{L}^{-1}A) < 1$, while this is not necessarily true of the incomplete inverse factors produced by biconjugation [15].

Let us finally note that in particular the process as modified in [16] will not break down for symmetric positive matrices. The modified method is known as *SAINV*. Indeed, in theory, AINV may suffer breakdown when the coefficient matrix is not an H-matrix.

3.1. Algorithmic variants

The procedure in Algorithm 5 is a *right looking* variant: when a vector z_i is finalized, it is used to update all the vectors $z_j, j > i$. An alternative formulation is to use a *left looking* variant: all the updates to z_i involving $z_j, j < i$, are performed in a single iteration of the outer loop; the relevant procedure is shown for Z in Algorithm 6 (the other triangle W can be handled in the same way). While the numerical behavior of the two algorithms is the same (in exact arithmetic), the distribution of work in the two variants is quite different. The left-looking variant groups together all the updates to a given column; it tends to perform more (sparse) dot products, using the "true" z_i (i.e. before sparsification), as it can afford to sparsify each column only once. These features have the following interesting properties that may be beneficial from a numerical point of view:

1. The dot products at 5 and 8 in Algorithm 6 are computed with the full vector z_i, before the application of the dropping tolerance;
2. The dropping rule on z_i entries is only applied at the end of the update loop, whereas in the right-looking version it would be applied at each update, thereby allowing a better buildup of the vector entries.

In our test problems the left-looking variant has suffered less from pivot breakdown.

3.2. Approximate biconjugation implementation

In Section 2.3 we have seen how incomplete factorization with inversion and sparsification can be implemented employing data structures and operators for partially ordered sets. We now direct our attention to the implementation of approximate inversion with biconjugation. It turns out that the scheme we are interested in can also be based on the same abstract data type.

Let us take a close look at Algorithm 6. In an actual implementation the vector z_i would be stored in full format during the execution of loop 4, and there could be two applications of a dropping rule:

1. at statement 6 the update of z_i is only performed for a sufficiently large value of p_i/p_j;
2. after the statement 8 a dropping rule is applied to z_i thereby sparsifying it for final storage.

Algorithm 6 Left Looking Biconjugation for Z

1: $z_1^{(0)} \leftarrow e_1$; $\quad p_1^{(0)} \leftarrow a_{11}$
2: **for** $i = 2, \ldots, n$ **do**
3: $\quad z_i^{(0)} \leftarrow e_i$
4: $\quad$ **for** $j = 1, \ldots, i-1$ **do**
5: $\quad\quad p_i^{(j-1)} \leftarrow a_j^T z_i^{(j-1)}$;
6: $\quad\quad z_i^{(j)} \leftarrow z_i^{(j-1)} - \left(\frac{p_i^{(j-1)}}{p_j^{(j-1)}}\right) z_j^{(j-1)}$
7: $\quad$ **end for**
8: $\quad p_i^{(i-1)} \leftarrow a_i^T z_i^{(i-1)}$;
9: **end for**

Note that the application of the first dropping rule based on p_i/p_j was actively discouraged in the paper that introduced right-looking AINV for symmetric systems [17].

In our experiments with the left-looking variant we have instead applied the dropping rule, without adverse numerical effects while at the same time providing a performance advantage. Moreover, in the dropping rule applied to z_i we apply the usual threshold comparison but we also enforce a limitation on the maximum number of nonzeros allowed, similarly to what happens in the $ILU(T, P)$ algorithm.

A key observation is that the execution of statement 5 in Algorithm 6 will compute the dot product among a_j and z_i even if in most cases this product will be exactly zero because of the (mis)match between the position of nonzeros in a_j and z_i; this is a useless quadratic cost. We can instead ensure that the loop 4 is executed only as necessary, i.e.: *we should only execute the iterations of the loop on j where the dot product at 5 is nonzero.* This is equivalent to letting at each step j be the lowest index among those not processed yet such that row a_{j*} has at least one nonzero element in a column corresponding to a nonzero entry in $z_i^{(j-1)}$. To achieve this goal we keep an extra copy of the pattern of a in a column-oriented format, and we do the following:

1. at the start of loop 4, $z_i \leftarrow e_i$; therefore, the set of indices $\{j\}$ is initialized with $R_{A_{*j}} = \{i : a_{ij} \neq 0\}$, the set of row indices of nonzero entries in column i of matrix A;
2. at each iteration of loop 4, choose j to be the smallest index in the set that is greater than the last visited one;
3. at step 6, whenever an entry $z_i(k)$ becomes nonzero, add the row indices $R_{A_{*k}}$ corresponding to the nonzeros of column k in matrix a to the set of indices to be visited.

To ease the implementation of the algorithm, we keep copies of the input matrix A both in a row-oriented and column-oriented storage. Having an extra copy of A in column-oriented format allows to build Z and W at the same time, sharing the same outer loop: when dealing with W we need to access rows and columns of A^T, but these are accessible as the columns/rows (respectively) of A. The inner loop is in any case separate between Z and W, as it runs on a subset of indices specific to the given triangular factor.

The result is Algorithm 7. The implementation makes use of a dense work vector zw to compute z_i and w_i; the indices of the non-zero entries are kept in a heap *hp*. Another heap *rhp* is used to hold the indices of the rows with at least one nonzero in a column matching a nonzero entry in zw, thus giving the set of rows for which we have to compute the scalar products.

We are now in a position to state:

Theorem 2. *Let* nz_a *be the average number of nonzeros per row in A and similarly* nz_z*; assume moreover the bounds*

$$|S| \leq \gamma \mathrm{nz}_a, \tag{11}$$
$$\mathrm{nz}_z \leq \beta \mathrm{nz}_a, \tag{12}$$

where $|S|$ *is the maximum cardinality of the sets of entries in any of the* z_i *before the application of the drop rule* 19*.Then the cost of Algorithm* 7 *is*

$$O(\gamma n \mathrm{nz}_a^2(1 + \beta(1 + \log(\gamma \mathrm{nz}_a)))). \tag{13}$$

Proof. Given the bound (11), the term $\gamma n \mathrm{nz}_a$ follows easily from the nesting of the two outer loops. The bodies of loops 8 and 23 in Algorithm 7 contain the following terms:

- The dot products at 10 and 25 add a term nz_a;
- The cost of 12 and 27 is given by $\beta \mathrm{nz}_a$, by assumption (12);
- The updates of the set S at 14 and 29 add another cost $\beta \mathrm{nz}_a \log(\gamma \mathrm{nz}_a)$.

The result follows considering that the statements 12–14 and 27–29 are executed at most as many times as statements 10 and 25 respectively, because of the drop rule at 11 and 26.

Algorithm 7 Practical left-looking biconjugation

1: For a matrix A let A_{i*} be the ith row, and A_{*j} the jth column;
2: For a sparse matrix A let $C_{A_{i*}} = \{j : a_{ij} \neq 0\}$ be the set of column indices in row i, and similarly let $R_{A_{*j}} = \{i : a_{ij} \neq 0\}$;
3: For a set S with an order relation $\leq$, let first(S) be the operator returning the smallest element in S;
4: $z_1^{(0)} \leftarrow e_1$; $p_1^{(0)} \leftarrow a_{11}$
5: **for** $i = 2, \ldots, n$ **do**
6: Inner loop over Z_j;
7: $zw \leftarrow e_i$; $S \leftarrow R_{A_{*i}}$;
8: **while** $S \neq \emptyset$ **do**
9: $j \leftarrow$ first(S); $S \leftarrow S - \{j\}$;
10: $p(i) \leftarrow A_{j*} zw$; $\alpha \leftarrow (p(i)/p(j))$;
11: **if** $|\alpha| > \epsilon$ (drop rule) **then**
12: $zw \leftarrow zw - \alpha Z_{*j}$
13: **for** $k \in R_{Z_{*j}}$ **do**
14: $S \leftarrow S \cup \{l \in R_{A_{*k}} : j < l < i\}$
15: **end for**
16: **end if**
17: **end while**
18: $p(i) \leftarrow A_{i*} zw$;
19: Apply a drop rule to zw;
20: $Z_{*i} \leftarrow zw$;
21: Inner loop over W_j;
22: $zw \leftarrow e_i$; $S \leftarrow C_{A_{i*}}$;
23: **while** $S \neq \emptyset$ **do**
24: $j \leftarrow$ first(S); $S \leftarrow S - \{j\}$;
25: $q(i) \leftarrow A_{*j}^T zw$; $\alpha \leftarrow (q(i)/q(j))$;
26: **if** $|\alpha| > \epsilon$ (drop rule) **then**
27: $zw \leftarrow zw - \alpha W_{*j}$
28: **for** $k \in R_{W_{*j}}$ **do**
29: $S \leftarrow S \cup \{l \in C_{A_{k*}} : j < l < i\}$
30: **end for**
31: **end if**
32: **end while**
33: $q(i) \leftarrow (A_{*i})^T zw$;
34: Apply a drop rule to zw;
35: $W_{*i} \leftarrow zw$;
36: **end for**

The situation is thus analogous to that of Theorem 1. To be completely precise, note that the bound β in (12) refers to the ratio between the size of the rows in the upper triangle Z and the rows in matrix A. When enforcing a size of the preconditioner comparable to that of the matrix A, the actual value of β will be approximately one half as that of the factor entering (7), since in that case we are comparing the upper triangle of the inverse to the upper triangle of the incomplete factorization. On the other hand, the biconjugation process is applied twice, for both Z and W, so that the ratio of nonzeros in the complete preconditioner to the nonzeros in A is again β just like in the case of *INVT*.

The application of the dropping rules at statements 11, 19, 26 and 34 of Algorithm 7 has the effect of enforcing strict control over the size of set S, thereby improving the factor γ and the overall performance of the preconditioner construction. A key element here is the fact that with dropping rules 19 and 34 we limit the number of accepted nonzeros.

The original AINV algorithm proposed in [17] may suffer from pivot breakdown when applied to matrices that are not H-matrices. In [16] a more robust version called SAINV is proposed: the key issue identified is the need to compute the diagonal elements p_i via the formula

$$p_i \leftarrow z_i^T A z_i,$$

instead of the simplified formula

$$p_i \leftarrow A_{i*} z_i.$$

The same kind of reasoning can be applied to nonsymmetric matrices, but we expect to reap less benefits, because in general the matrix A does not necessarily define a dot product. On the other hand, there are important cases where pivot breakdown cannot occur for SAINV also in the nonsymmetric case, in particular, if the symmetric part of the matrix is positive definite [18].

If we apply the full formula to the left-looking algorithm we obtain Algorithm 8: the product with A is applied at steps 10, 24 and 34. Note that the two triangles W and Z are no longer independent of each other: the computation of the p_i and

Algorithm 8 Practical left-looking biconjugation stabilized

1: For a matrix A let A_{i*} be the ith row, and A_{*j} the jth column;
2: For a sparse matrix A let $C_{A_{i*}} = \{j : a_{ij} \neq 0\}$ be the set of column indices in row i, and similarly let $R_{A_{*j}} = \{i : a_{ij} \neq 0\}$;
3: For a set S with an order relation $\leq$, let first(S) be the operator returning the smallest element in S;
4: $z_1^{(0)} \leftarrow e_1$; $p_1^{(0)} \leftarrow a_{11}$
5: **for** $i = 2, \ldots, n$ **do**
6: Inner loop over Z_j;
7: $zw \leftarrow e_i$; $S \leftarrow R_{A_{*i}}$;
8: **while** $S \neq \emptyset$ **do**
9: $j \leftarrow$ first(S); $S \leftarrow S - \{j\}$;
10: $p(i) \leftarrow ((W_{*j})^T A)zw$; $\alpha \leftarrow (p(i)/p(j))$;
11: **if** $|\alpha| > \epsilon$ (drop rule) **then**
12: $zw \leftarrow zw - \alpha Z_{*j}$
13: **for** $k \in R_{Z_{*j}}$ **do**
14: $S \leftarrow S \cup \{l \in R_{A_{*k}} : j < l < i\}$
15: **end for**
16: **end if**
17: **end while**
18: Apply a drop rule to zw;
19: $Z_{*i} \leftarrow zw$;
20: Inner loop over W_j;
21: $zw \leftarrow e_i$; $S \leftarrow C_{A_{i*}}$;
22: **while** $S \neq \emptyset$ **do**
23: $j \leftarrow$ first(S); $S \leftarrow S - \{j\}$;
24: $q(i) \leftarrow (AZ_j)^T zw$; $\alpha \leftarrow (q(i)/q(j))$;
25: **if** $|\alpha| > \epsilon$ (drop rule) **then**
26: $zw \leftarrow zw - \alpha W_{*j}$
27: **for** $k \in R_{W_{*j}}$ **do**
28: $S \leftarrow S \cup \{l \in C_{A_{k*}} : j < l < i\}$
29: **end for**
30: **end if**
31: **end while**
32: Apply a drop rule to zw;
33: $W_{*i} \leftarrow zw$;
34: $p(i) \leftarrow q(i) \leftarrow (W_{*i})^T AZ_{*i}$;
35: **end for**

q_i must be performed at step 34 where we finally have available the relevant elements of both W and Z. In the test set used in this paper we have not found any significant advantage in using Algorithm 8 over the use of Algorithm 7.

4. Approximate inverses: Algorithmic variants

We now discuss some features of the algorithms presented.
For reference we denote by:

INVK: the Algorithm 2 based on the sparse inversion of triangular factors based on a positional drop strategy;
INVT: the Algorithm 4 based on the sparse inversion of a triangular factors based on a numerical drop strategy;
LLK: the Algorithm 7 based on a left-looking variant of AINV biconjugation.

These different algorithms have different implementation costs. Moreover, a discussion of their cost based on the number of floating-point operations only is likely to be misleading.

4.1. Inversion of sparse ILU factors

The inversion of underlying sparse ILU factors, *INVK* and *INVT* variants, can be implemented with the same building blocks that have been used for $ILU(K)$ and $ILUT(T, P)$ incomplete factorizations, respectively. The above inversion algorithms require the choice of multiple parameters, and are therefore somewhat difficult to tune in actual applications. In the case of *INVK* it is necessary to choose the level of fill in the sparse factorization and the level of additional fill in the approximate inversion phase: $INVK(N_1, N_2)$ means having an $ILU(N_1)$ factorization, and then an inversion accepting fill-in for N_2 levels beyond the output of the factor phase. Thus, an $INVK(0, 0)$ preconditioner has exactly the same pattern as the original matrix A.

In general it is better to allow additional fill in the inversion phase rather than in the factorization phase; this is perhaps not surprising considering that the inverse of an irreducible sparse matrix is usually full, therefore we need to spend space resources (i.e. nonzeros) to approximate more closely the sparse triangular factor, taken as a reference in terms of preconditioning efficiency.

For *INVT*, similar considerations apply, except that we have to choose four parameters: the drop threshold ϵ and the number of additional nonzeros N for both the incomplete factorization and the sparse inversion. Again, the sparse inversion phase is based on the kernels developed for the incomplete factorization; this in turn defines a dual drop strategy based on the numerical threshold and on accepting at least as many nonzero entries as in the original matrix A, and up to N additional nonzeros, with N chosen by the user.

We will henceforth denote this algorithm as $INVT(N_1, \epsilon_1, N_2\epsilon_2)$ where the $\epsilon_{1,2}$ are the thresholds for the factor and inversion steps, respectively. The parameters N_1 and N_2 specify the factor β implicitly: they are interpreted as the maximum number of nonzeros per row to be accepted in output in addition to those already present in input. Thus, $INVT(0, \cdot, 1, \cdot)$ would accept at most as many nonzeros as in A in the factor phase, and at most one more per row per triangular factor in the inversion phase.

We can derive from Theorem 1 and from the performed tests useful suggestions for the choice of the algorithmic parameters:

- as a first approximation, the performance of the final preconditioner is related to the number of its nonzeros (although this is not guaranteed to be always true). As we let β grow, we typically get a better approximation to the inverse, so the number of iterations decreases, but the cost of the preconditioner, both in terms of building and applying it, grows. Often a value of β only moderately larger than 1 suffices;
- The threshold employed at statement 6 of Algorithm 4 controls the size of the set S: having too permissive a threshold is wasteful, since many nonzeros will be kept around, only to be thrown away when sparsifying $\hat{u}_i$ at step 13. Thus the threshold should be as restrictive as possible while still providing enough nonzeros to match the desired β.

4.2. Biconjugation

The implementation of our left-looking variant of biorthogonalization requires the choice of two parameters: the dropping threshold ϵ and the amount of fill-in p. We stress that for non symmetric or non Hermitian matrices we speak about *biorthogonalization* and *conjugation* for symmetric or Hermitian ones.

We always use the same threshold and fill-in for the Z and W factors, even if in principle they could be different. The construction of each factor is logically independent of the other; nevertheless, we compute them simultaneously, making maximal use of both a row-oriented and a column-oriented copy of A. The left-looking incomplete biconjugation will be henceforth denoted as $LLK(N, \epsilon)$, where ϵ is the dropping tolerance and N is the number of nonzeros per row in each triangular factor; thus, the effective number of nonzeros per row in the preconditioner will be determined by $2N + 1$.

Comparing Eqs. (8) and (13), we see that the computational complexity bounds for *INVT* and *LLK* are of the same order. This is substantiated by the numerical results in Section 6 where the ratio of the computing time of the two preconditioner types is approximately constant once the method(s) parameters are adjusted to give preconditioners with comparable number of nonzeros, and with comparable intermediate sizes during the factorization/biconjugation loops. The advantage of *LLK* is that it is normally easier to adjust the control parameters of the algorithm to achieve the desired result. On the other hand, once the parameters are tuned, we experienced that the build phase of the *INVT* and *INVK* preconditioners is often faster.

4.3. Reordering: numerical and algorithmic implications

Reordering is the process of applying a permutation to both rows and columns of a sparse matrix. If P is the permutation, then the new matrix is $\hat{A} = PAP^T$.

Also nonsymmetric permutations $\hat{A} = PAQ^T$, $Q \neq P$, have been shown to be very beneficial in many cases but will not be treated here.

The effects of reordering on preconditioners are the subject of a substantial body of research, even though the situation is far from being fully clear. Both factorization-based and inversion-based preconditioners have been analyzed in the past; see, e.g., [19,11,12,20].

A recurring theme in sparse matrix computations is the relative weight of data structure manipulations and other integer operations as opposed to the "true" floating point arithmetic operations; both are affected by reorderings. In particular it is often observed that

- the amount of fill during the computation of a factorized approximate inverse strongly depends on the ordering of the matrix;
- the interaction with the computer memory subsystem, and the reduction in operations due to "clipping" techniques applied to the matrix profile, tend to favor orderings designed to reduce bandwidth, such as *Reverse Cuthill–McKee* (or RCM for short).

Orderings developed for sparse factorization algorithms such as *Nested Dissection* and *Minimum Degree* give in general lower fill-in, therefore better approximation for a fixed amount of allowed fill-in; Benzi and Tůma prove the following (see [12])

Theorem 3. *Consider a matrix A arising from a five-point discretization of an elliptic PDE on a two-dimensional regular $k \times k$ grid. Then, the number of nonzeros in the inverse factor L^{-1} is $O(k^3)$ for Nested Dissection and $O(k^4)$ for Reverse Cuthill–McKee orderings, respectively.*

And yet the situation is not quite as bad as it sounds; in the same paper [12], by using the main result in [21], the authors derive a bound for the entries of the factors of the approximate inverses.

Theorem 4. *Let A be SPD and m-banded, with $\max_i a_{ii} = 1$; then the entries z_{ij}, $i < j$ in $Z = L^{-T}$, are bounded by*

$$|z_{ij}| \leq K\lambda^{j-i} \tag{14}$$

for appropriate K and λ.

The actual definition of λ is related to the spectral condition number κ, and is given by

$$\lambda = q^{2/m}, \qquad q = q(\kappa) = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}.$$

The most important feature entailed by Theorem 4 is by far the fact that for important applications, even though the inverse matrix is full, its *entries* decay away from the main diagonal. The value of λ determines how fast the entries decay; this property becomes computationally interesting when λ is not too close to one. Moreover, K also depends on the condition number and on the bandwidth of A and can be written explicitly in terms of these quantities; see [12].

As a consequence, using a bandwidth-reducing ordering appears to be beneficial. Benzi and Tůma as well as Bridson and Tang [20] conclude that for the preconditioners based on biconjugation, fill-reducing orderings give better results in general.

5. Software issues and computing platforms

The study of the multiple variants of approximate inverse preconditioners discussed here required the development of a substantial amount of software. A complete and detailed discussion of the software development issues involved is outside the scope of this work.

The Krylov subspace iterative methods and the basic kernels for computing matrix–vector products are those of the Parallel Sparse BLAS (PSBLAS) library [22]. This software library has the ultimate goal of enabling the implementation of Krylov methods on distributed memory computing architectures employing the MPI programming interface. To achieve this, it was necessary to develop a number of support tools for handling sparse matrices in serial mode, both in terms of support operations and of computational kernels. The library has been subject to a major redesign employing object-oriented techniques, described in [23] and resulting in its version 3.0. The main objective for the redesign was to make it easy for developers to evolve the library over time by adding plugins for new computing architectures without the need to change the main library framework. This capability has been exploited to implement in a convenient way a plugin to use the computing capability of graphics processing units, commonly known as GPUs (see also Section 5.1). The specific techniques employed involve the usage of the STATE and PROTOTYPE design patterns [24], which may be described in a nutshell as follows:

1. The STATE pattern consists of the encapsulation of an object in a two-layered hierarchy, so that the inner object can be changed dynamically to adapt to the various usage needs of the application.
2. The PROTOTYPE pattern tackles a problem for a library developer: the library code must be able to instantiate at run time objects whose type is not known (exactly) at the time of writing and compiling. In particular, the object the library needs to instantiate is of a type that extends a known type, but was developed after the main library was written and compiled, e.g. to adapt to a new computing architecture. The solution is to make the library code use a sample object provided by the application code. In our case the application code will declare an object of a specific GPU-enabled type, and will pass this object to the library which will then instantiate a copy of it to run the necessary computations (e.g. matrix–vector products) on the GPU device.

A full discussion of these techniques has been detailed in [25,3].

Note that since the approximate inverse preconditioners work by applying sparse matrix–vector products, the necessary kernels and programming techniques are a by-product of the effort to implement Krylov methods, which also rely on the sparse matrix–vector product in their formulation. Since the matrix–vector product *per se* has already been discussed in other works, in this article we have mostly discussed the programming techniques that have been applied to the construction of the preconditioner matrices.

The preconditioners themselves have been implemented in the context of the MLD2P4 framework [26]. This is a package of multilevel preconditioners that can be plugged into the PSBLAS library in an easy and transparent way. In particular we refer to the current development version 2, since this uses PSBLAS 3.0.

As already mentioned, MLD2P4 version 2 is a framework supporting algebraic multilevel preconditioners. It supports a hierarchy of objects including "restriction" and "prolongation" operators that move between levels, "smoothers" that are applied to a given level using "solvers" applied to a given subdomain, according to the lexicon typical of algebraic multigrid. From a software point of view, approximate inverses are simply new variants of the inner "solver" objects that can be plugged into the general framework, using the same design patterns we discussed above.

For the purposes of this article we are only discussing the behavior of approximate inverses as preconditioners. We do not exploit the full generality of the multilevel preconditioner structure. Nevertheless, by construction the framework supports the usage of approximate inverses in more sophisticated ways. Some preliminary results are presented in [27].

We are testing some ideas to implement the construction phase of the underlying preconditioners on the GPU itself. At the present stage, it seems that the left-looking variant of biconjugation is less suitable than the right-looking to exploit the GPU computing features. Since we are still at a preliminary stage, we will only present here results related to building the preconditioners on the CPU. Note however that there is a positive aspect in that we have not tied down the approximate inverses to the GPU usage: they are two different plugins that can be used together or separately, thereby demonstrating the flexibility of the underlying software structure, and providing the ability to run the experiments on more conventional architectures.

5.1. Approximate inverses and computing devices: GPUs

In the context of the present article the main interest of GPUs is not only in their performance characteristics, but also in the algorithmic implications of their programming model. The parallelization needed to exploit the capabilities of these machines can be quite influential in the formulation of the models and algorithms for solving interesting application problems. The characteristics of the GPUs have been discussed in a number of papers, among them there are [28,4,3,2] which are relevant to sparse matrix computations. See also [29–31], where the latest focus on the construction phase of the preconditioners.

In particular, it should be stressed that the usage of GPUs has the effect of making approximate inverses more attractive with respect to the standard incomplete factorizations. In fact, there exist efficient implementations of a sparse matrix by vector product, whereas it is difficult to have an efficient direct solver or iterative method with incomplete factorization preconditioner for sparse linear system on the GPU. As an example, the conjugate gradient preconditioned with incomplete LU available in CUDA CuSPARSE since version 4.0 [32] achieves at best a speedup factor of about 2, whereas approximate inverses give much better results as we will detail in Section 6.

Finally, we note that the NVIDIA GPUs do *not* have virtual memory: memory occupation must be managed by the application. This makes it desirable to have fine-tuned control over the size of the preconditioning operators.

6. Numerical tests

We now report on some experiments on both synthetic and real-world applications. Even though the PSBLAS and MLD2P4 software frameworks discussed in Section 5 are based on MPI, for the current set of experiments we report runs with just one MPI process. Further testing in the context of multilevel preconditioners with multiple MPI processes is the subject of related work, to be fully explored in the future. The computing platform used is based on an Intel Xeon E5-2670 running at 2.6 GHz, coupled with an NVIDIA K20M graphics accelerator. The GPU kernels have been compiled with CUDA 6.5. All other software components have been built with the GNU compilers (C and Fortran) version 4.8.3. Symmetric linear systems are solved with the CG method, whereas non symmetric systems are solved with BiCGSTAB [33]; with a stopping criterion based on the reduction of the 2-norm of the (left preconditioned) relative residual, and a stopping tolerance of 10^{-7}.

Table 2 shows tests run completely on the CPU. All other measurements refer to a preconditioner build phase run on the CPU and a linear system solution run on the GPU.

6.1. Convection–diffusion in 2D and 3D

We start with some tests based on the linear convection–diffusion PDE model problem

$$-\nu\nabla\cdot(a(x)\nabla u)+q(x)\cdot\nabla u=0,\quad x\in\Omega, \tag{15}$$

$$u=g,\qquad x\in\partial\Omega, \tag{16}$$

where Ω is $[0,1]^d$, d is the dimension, here $d=2$ or $d=3$. We use standard second order centered finite difference discretization with constant step size. Note that bounds for the spectrum of the eigenvalues of the underlying system of linear equations, i.e., of the discretized model, are known. Moreover, for some preconditioners, bounds for the preconditioned spectrum and the condition number of the related matrix of the eigenvectors are known under certain simplifying assumptions, see, e.g., [34]. Different choices for the function $a(x)$ are possible. We have run tests with $a(x)=1$, $x=(x_1,x_2,\ldots)^T$, and $a(x)=\exp(-\sum_i x_i)$, the second case being harder than the first case for the linear solvers because the considered domain includes the origin. The parameter ν controls the relative strength of the diffusion and convection parts of the equations, and indirectly the conditioning of the resulting linear system. The tests have been run with $\nu=1/80$. We have built both

Table 1
Linear system sizes—PDE test cases.

2D		3D	
Matrix	Size	Matrix	Size
pde0300	90,000	pde020	8,000
pde0400	160,000	pde030	27,000
pde0500	250,000	pde040	64,000
pde0600	360,000	pde050	125,000
pde0700	490,000	pde060	216,000
pde0800	640,000	pde070	343,000
pde0900	810,000	pde080	512,000
pde1000	1,000,000	pde090	729,000
pde1100	1,210,000	pde100	1,000,000

Table 2
Baseline CPU performance data, $a(x) = 1$, CG.

Matrix	NOPREC		*ILU*(0)			*INVK*(0, 0)		
	it	tslv	tpr	it	tslv	tpr	it	tslv
pde0300	728	0.69	0.1	133	0.31	0.1	375	0.75
pde0400	960	1.68	0.2	175	0.75	0.2	495	1.88
pde0500	1189	3.52	0.2	216	1.46	0.4	614	3.93
pde0600	1417	7.04	0.3	256	2.62	0.5	731	6.91
pde0700	1643	10.30	0.4	296	3.95	0.7	847	10.75
pde0800	1867	16.00	0.5	336	5.92	0.9	962	15.66
pde0900	† 2000	–	0.6	376	8.36	1.2	1075	21.91
pde1000	† 2000	–	0.7	416	11.54	1.5	1189	29.53
pde1100	† 2000	–	0.9	455	15.05	1.8	1301	39.38

Table 3
Performance data for *ILU*(0) on GPU, $a(x) = 1$, CG.

Matrix	*ILU*(0) CPU			*ILU*(0) GPU CSR		*ILU*(0) GPU HYB	
	tpr	it	tslv	it	tslv	it	tslv
pde0300	0.097	133	0.31	133	2.14	133	0.31
pde0400	0.152	175	0.75	175	3.94	175	0.74
pde0500	0.216	216	1.46	216	6.25	216	1.46
pde0600	0.292	256	2.62	256	9.17	256	2.59
pde0700	0.379	296	3.95	296	12.56	296	3.94
pde0800	0.483	336	5.92	336	16.86	336	6.02
pde0900	0.599	376	8.36	376	21.63	376	8.31
pde1000	0.722	416	11.54	416	27.27	416	11.36
pde1100	0.862	455	15.05	455	33.83	455	15.24

2D and 3D variants of this equation. The names of the test cases contain the number of steps in each coordinate direction for unit square and cube, respectively. The resulting linear system sizes are detailed in Table 1. In all tables the † symbol marks those cases where convergence was not reached within the allowed number of iterations.

For the first set of tests we have used the natural numbering scheme, i.e. no reorderings have been applied. The baseline performance for our analysis is shown in Table 2. Here we show measurements gathered running both the preconditioner setup and the Krylov solver on the CPU; they refer to the 2D equation with no preconditioning, *ILU*(0) and *INVK*(0, 0). From this comparison it looks like *INVK*(0, 0) is not a very attractive option. It is substantially underperforming when compared with *ILU*(0). Indeed, when the method converges without a preconditioner within the allowed iterations, it is faster even if the iteration count is much worse. This is far from surprising, since the *INVK*(0, 0) preconditioner starts from *ILU*(0) and then it computes its inverse in an approximated manner, therefore the increase in the amount of work per iteration is not compensated by a sufficient reduction in the iteration count.

If we perform the same tests on the GPU, the results are quite different. The *ILU* preconditioners rely on the solution of sparse triangular linear systems; the triangular structure enforces data dependencies, and the sparsity of the matrix reduces the amount of floating-point operations. The GPU architecture employs a large number of relatively slow arithmetic units; to use them effectively, we need to feed them with a significant amount of independent computations, and this is extremely difficult in the context of a sparse triangular system solution.

In Table 3 we show some performance data obtained with an implementation of *ILU*(0) based on the data storage formats CSR and HYB available in the NVIDIA CUSPARSE library version 6.5. The first set of data in the second, third and fourth columns is just a repetition of the CPU data from the previous table. The second set of measurements in the fifth and sixth columns uses the CSR format: we actually get a *slowdown*, even a significant one. When using HYB the GPU timings are

Table 4
Basic GPU performance data, 2D test case, $a(x) = 1$, CG.

Matrix	NOPREC		*INVK*(0, 0)			*INVT*(0, .1, 4.01)		
	it	tslv	tpr	it	tslv	tpr	it	tslv
pde0300	728	0.69	0.1	375	0.58	0.6	286	0.47
pde0400	960	0.94	0.2	495	0.83	1.0	377	0.71
pde0500	1189	1.20	0.4	614	1.11	1.5	468	1.09
pde0600	1417	1.51	0.5	731	1.45	2.2	556	1.31
pde0700	1643	2.00	0.7	847	1.93	3.0	646	1.85
pde0800	1867	2.59	0.9	962	2.50	3.9	732	2.25
pde0900	† 2000	–	1.2	1075	3.23	4.9	820	2.78
pde1000	† 2000	–	1.5	1189	4.20	6.0	906	3.59
pde1100	† 2000	–	1.8	1301	5.18	7.5	994	4.63

Table 5
Basic GPU performance data, 2D test case, $a(x) = \exp(-(x_1 + x_2))$, CG.

Matrix	NOPREC		*INVK*(0, 0)			*INVK*(0, 1)		
	it	tslv	tpr	it	tslv	tpr	it	tslv
pde0300	1446	0.43	0.1	408	0.64	0.2	310	0.50
pde0400	1926	0.89	0.2	538	0.90	0.3	410	0.71
pde0500	† 2000	–	0.4	667	1.20	0.5	509	0.98
pde0600	† 2000	–	0.5	795	1.57	0.8	606	1.40
pde0700	† 2000	–	0.7	922	2.09	1.0	703	1.79
pde0800	† 2000	–	0.9	1048	2.71	1.3	799	2.41
pde0900	† 2000	–	1.2	1174	3.50	1.7	895	3.03
pde1000	† 2000	–	1.4	1299	4.49	2.1	991	3.93
pde1100	† 2000	–	1.8	1424	5.76	2.5	1085	4.96

Table 6
GPU performance: *INVT* vs. *LLK* on 2D test case, $a(x) = 1$, CG.

Matrix	*INVT*(0, .1, 4, .01)				*LLK*(5, 0.01)			
	tpr	it	tslv	spd	tpr	it	tslv	spd
pde0300	0.6	286	0.47	1.6	0.3	289	0.46	1.6
pde0400	1.0	377	0.71	2.6	0.5	380	0.69	2.6
pde0500	1.5	468	1.09	3.5	0.8	471	0.90	4.1
pde0600	2.2	556	1.31	4.8	1.2	561	1.20	5.1
pde0700	3.0	646	1.85	5.5	1.6	650	1.64	5.8
pde0800	3.9	732	2.25	6.6	2.1	739	2.10	6.7
pde0900	4.9	820	2.78	7.6	2.7	826	2.75	7.2
pde1000	6.0	906	3.59	8.0	3.3	913	3.51	7.7
pde1100	7.5	994	4.63	8.2	4.0	996	4.47	8.0

essentially the same as the CPU times, despite the fact that the matrix–vector product in HYB format is significantly faster than the CPU one. From the NVIDIA documentation we surmise that the library is applying some sort of level numbering to the triangular matrices, but the amount of parallelism that can thus be extracted is apparently too low in many practical cases to make good use of the GPU capabilities. Slightly better results can be obtained in the case of 3D problems, but we have never seen a speedup over 1.5, and Naumov reports in [32] a best speedup of about 2.

In Table 4 we see the results from the application of approximate inverses:

- The preconditioned iterations always converge within the allowed number of iterations;
- The solve times are as good as or better for the preconditioned iterations with respect to the unpreconditioned ones;
- The solve times for *INVK* and *INVT* are very similar (with INVT better), but the time for building the preconditioner is quite different;
- The speedup of the GPU over CPU on iterations preconditioned with the approximate inverse is quite substantial; enough, indeed, to overcome the advantage of *ILU*(0) in the number of iterations.

A similar situation is shown in Table 5 for the harder test case with

$$a(x) = \exp(-(x_1 + x_2));$$

here the unpreconditioned iterations do not converge at all, and we need to increase β to reach convergence. Note that the reduction in the number of iterations when going from *INVK*(0, 0) to *INVK*(0, 1) does not translate into the same reduction in runtime, because the memory footprint of the preconditioner increases significantly.

In Tables 6 and 7 we have a comparison with the symmetric left-looking *LLK* variant. For the solution phase it is the fastest option; for the setup phase it is faster than *INVT*, although slower than *INVK*, and it works consistently well, as opposed to the

Table 7
GPU performance: *INVK* vs. *LLK* on 2D test case, $a(x) = \exp(-(x_1 + x_2))$, CG.

Matrix	*INVK*(0, 1)				*LLK*(10, 0.05)			
	tpr	it	tslv	spd	tpr	it	tslv	spd
pde0300	0.2	310	0.50	1.6	0.4	242	0.42	1.9
pde0400	0.3	410	0.71	2.9	0.7	321	0.62	3.3
pde0500	0.5	509	0.98	4.2	1.1	397	0.91	5.3
pde0600	0.8	606	1.40	5.0	1.6	472	1.33	5.1
pde0700	1.0	703	1.79	6.0	2.2	549	1.62	7.0
pde0800	1.3	799	2.41	6.6	2.8	623	2.17	8.1
pde0900	1.7	895	3.03	7.5	4.1	698	2.83	7.9
pde1000	2.1	991	3.93	7.8	4.3	772	3.58	9.5
pde1100	2.5	1085	4.96	8.3	5.4	845	4.63	9.9

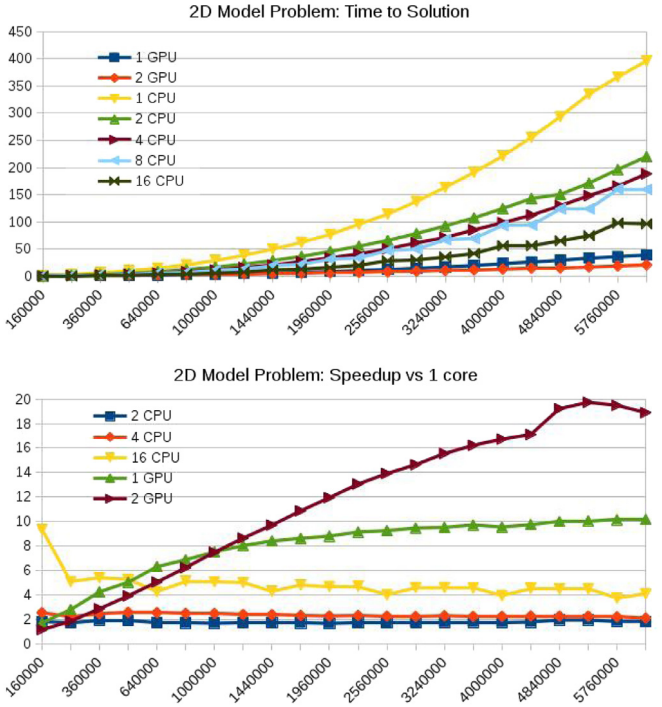

Fig. 1. Parallel performance data, 2D test case, $a(x) = 1$, CG.

occasional problems of *INVK*. The behavior in terms of number of iterations to convergence is consistent with the theoretical expectations, i.e., they are roughly proportional to n. We also list speedups with respect to running the same algorithms on the CPU, which for large enough matrices are about an order of magnitude. We also have a non-trivial speedup of about 3.4 when comparing against *ILU*(0) on CPU, despite the fact that the number of iterations is larger for approximate inverses. We stress that *ILU*(0) would provide no speedup at all with a GPU implementation.

Note that *INVK*(N_1, N_2) has a rather significant disadvantage with respect to the other alternatives, in that the control over the number of nonzeros in the preconditioner is exercised through the fill levels, and therefore it is at a much coarser grain.

6.2. Multi-core and multi-GPU performance

The programming framework of PSBLAS/MLD2P4 is fully parallel; therefore we may choose to run our application by using multiple processes. Here we show the results of two tests, the basic 2D and 3D convection–diffusion problem, preconditioned with one sweep of Block Jacobi using INVT as the local solver on both CPU and GPU. The tests were run on 1 or 2 GPUS, and on 1–16 CPUs, within one shared-memory node. From the graphs in Figs. 1 and 2 we can see that for these problems even a single GPU is capable of performing at more than twice the speed of 16 cores acting together, and at sufficiently large sizes we also have a good speedup from using two GPUs to solve the linear system; the solution times for the largest 2D cases are about 100 s for the 16-cores, 48 s for single GPU and 23 for dual GPUs. This behavior is consistent with the fact that the sparse matrix–vector product is a memory-bound kernel, therefore its performance is determined

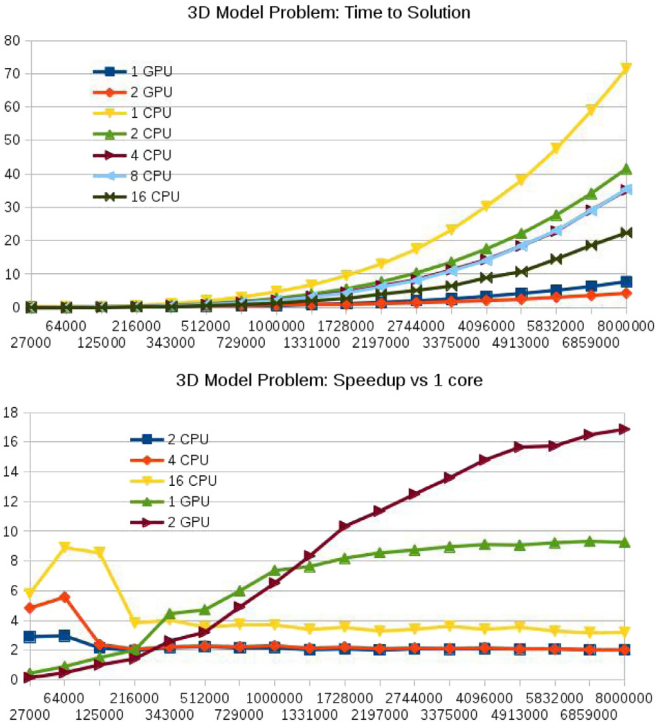

Fig. 2. Parallel performance data, 3D test case, $a(x) = 1$, CG.

by the available memory bandwidth. On our experimental platform the CPU cores are divided in two 8-core banks, each having a peak bandwidth of 51 GB/s, whereas each K20 GPU has a peak memory bandwidth of 208 GB/s. One thing to notice is that there is a break-even point between the solution times with one and two GPUs, and this is approximately located at a number of equations of 10^6; all these performance data are consistent with the behavior in the image segmentation application examined in [35]. At (very) small sizes the multi-core versions exhibit spikes in the speedup graphs. These are due to the fact that the smallest matrices are small enough to fit into the combined cache memories of the 8/16 cores.

A more detailed analysis of the behavior of the approximate inverse preconditioners in a parallel context would require looking at the interaction with multilevel aggregation, and the use of heterogeneous local solvers (factorization and approximate inverses) on CPU cores and GPUs, as easily enabled by our software framework. Such a complete analysis is beyond the scope of this article; some preliminary data appeared in [27], and a more thorough investigation taking into account recent hardware developments is the subject of ongoing work.

6.3. Engine design application

This test set was extracted from an engine simulation application [36,37]. The latter solves the turbulent Navier–Stokes equations with ALE finite volume discretization, coupled with the k–ϵ turbulence model and solved with the SIMPLE method. In particular, the test matrices are those for the pressure-correction equation, which is the most demanding linear system to be solved in this application. The coefficient matrices are non-symmetric, but with a symmetric sparsity pattern, and they have no more than 19 nonzero entries per row. They are obtained from the simulation of the cycle of a Diesel engine, at various positions of the piston inside the cylinder; the extreme cases (kivap1 and kivap9) were extracted with the piston at the bottom center, while kivap5 is close to the top center.

In Table 8 we can see the behavior of the simplest preconditioners; near the top center the linear systems are significantly harder and both *INVK* and *INVT* methods suffer. From Table 9 we can extract the following considerations:

- With the appropriate parameters *INVT* can be quite competitive, sometimes even better than *LLK*;
- Finding the correct set of parameters for *INVT* is a non-trivial task;
- *LLK* is easier to tune; indeed, the fill factor of 10 is easy to derive as being roughly half of the average number of nonzeros in the test matrices;
- The speedups for the GPU version over the CPU are not too high, but they are in line with the results from the model test cases given the relatively small size of the matrices involved.

We should also note that in our experience, keeping with similar performance in terms of iterations, the number of nonzeros for the *LLK* preconditioner is often less than the corresponding number for *INVT*.

Table 8
GPU performance: engine design application.

Matrix	Size	NOPREC		*INVK* (0, 1)			*INVT* (0, .01, 1, .01)		
		it	tslv	tpr	it	tslv	tpr	it	tslv
kivap1	86304	95	0.320	2.040	21	0.117	1.230	24	0.108
kivap2	76504	204	0.430	1.760	49	0.241	1.240	64	0.251
kivap3	59354	257	0.686	1.340	56	0.264	1.000	84	0.334
kivap4	42204	636	1.259	0.940	138	0.592	0.670	135	0.343
kivap5	25054	† 1000	1.955	0.540	† 1000	2.218	0.310	† 1000	1.930
kivap6	42204	856	2.025	0.900	250	0.735	0.610	214	0.534
kivap7	56904	257	0.828	1.260	55	0.241	0.980	74	0.295
kivap8	76504	197	0.604	1.700	49	0.254	1.240	62	0.268
kivap9	86304	203	0.675	1.950	46	0.205	1.320	65	0.287

Table 9
GPU performance: engine design application, best case.

Matrix	Size	*LLK* (10, 0.01)				*INVT* (1, .01, 4, .01)			
		tpr	it	tslv	spd	tpr	it	tslv	spd
kivap1	86304	1.73	24	0.083	2.8	1.31	20	0.091	2.4
kivap2	76504	1.66	62	0.189	2.7	1.33	54	0.237	2.3
kivap3	59354	1.28	67	0.261	1.6	1.07	73	0.270	2.1
kivap4	42204	0.80	101	0.242	1.8	0.71	103	0.267	2.1
kivap5	25054	0.38	41	0.077	1.2	0.34	53	0.105	1.4
kivap6	42204	0.80	101	0.384	1.1	0.67	109	0.281	2.0
kivap7	56904	1.23	67	0.234	1.7	1.03	64	0.172	2.8
kivap8	76504	1.65	54	0.228	1.9	1.33	52	0.229	2.3
kivap9	86304	1.76	54	0.233	2.2	1.41	44	0.184	2.7

Table 10
UFL sparse matrix collection linear system sizes.

Matrix	M	NNZ	Matrix	M	NNZ
A_500k	531,612	2,629,578	thermal2	1,228,045	8,580,313
A_1M	995,100	4,892,218	thermomech_TC	102,158	711,558
ML_Laplace	377,002	27,689,972	nlpkkt80	1,062,400	28,704,672
bcsstk16	4,884	290,378	parabolic_fem	525,825	3,674,625
raefsky2	3,242	294,276	pde060	216,000	1,490,400
lung2	109,460	492,564	pde080	512,000	3,545,600
FEM_3D_thermal2	147,900	3,489,300	pde100	1,000,000	6,940,000
Poisson3Db	85,623	2,374,949			

6.4. University of Florida collection

To complete our experiments we have picked a subset of the matrices in the University of Florida Sparse Matrix Collection [38]. To this subset we have added the engine design matrices of the previous subsection, and five matrices from simple elliptic problems:

- The cases pde060, pde080 and pde100 are from the three-dimensional version of the convection–diffusion equation used previously;
- The cases A-500k and A-1M arise from a finite volume tetrahedral mesh discretization of the thermal diffusion in a solid (copper) bar.

In Table 10 we report the size of the linear systems.

Renumbering, or reordering, and its effect on preconditioners has been investigated by many authors. Some works relevant to this paper appeared previously in [11,39,12]. In these papers it is argued that a fill-reducing ordering is in general to be preferred. We have thus tested this with the *AMD minimum degree algorithm* [40,41] compared with the natural ordering and with the *Gibbs, Poole and Stockmeyer* variant (*GPS* in the tables) of *reverse Cuthill–McKee* numbering [42,43].

As it turns out, AMD is practically never beneficial as detailed in Tables 11 and 12. The effect of GPS is on average less than expected in our tests. In our opinion, this could be due to the differences in the memory hierarchy inside the GPU device with respect to normal CPU hierarchies.

Finally, we emphasize that the different effect of reorderings (such as GPS) for the (limited) tests we performed makes it difficult to give general statements on the interplay of sparse matrix orderings and preconditioning.

Table 11
Effect of renumbering on *INVT*, UFL collection.

Matrix	*INVT* (1, .01, 2, .01)								
	NONE			GPS			AMD		
	tpr	it	tslv	tpr	it	tslv	tpr	it	tslv
A_500k	4.32	69	0.482	4.87	85	0.435	5.05	81	0.666
A_1M	9.31	66	0.537	9.12	66	0.512	9.28	78	1.073
ML_Laplace	32.50	†	11.537	–	–	–	–	–	–
bcsstk16	0.31	34	0.065	0.31	37	0.072	0.30	38	0.074
raefsky2	0.27	53	0.099	0.28	53	0.100	0.28	53	0.121
lung2	0.68	†	3.839	–	–	–	0.50	†	3.133
FEM_3D_thermal2	2.48	7	0.046	2.69	6	0.041	3.08	6	0.044
Poisson3Db	3.14	75	0.327	2.51	72	0.362	2.89	60	0.336
thermal2	11.48	994	12.600	14.92	†	10.994	12.21	†	19.158
thermomech_TC	0.89	4	0.118	0.73	4	0.117	0.85	4	0.118
nlpkkt80	26.79	†	14.462	–	–	–	–	–	–
parabolic_fem	3.72	593	3.580	5.88	673	3.400	5.17	716	8.175
pde060	1.70	48	0.167	1.63	49	0.200	2.31	47	0.252
pde080	3.86	62	0.305	3.85	65	0.389	5.86	56	0.474
pde100	10.29	87	0.672	11.02	72	0.635	11.72	55	0.889

Table 12
Effect of renumbering on *LLK*, UFL collection.

Matrix	*LLK* (8, .01)								
	NONE			GPS			AMD		
	tpr	it	tslv	tpr	it	tslv	tpr	it	tslv
A_500k	6.96	71	0.482	6.33	89	0.511	10.47	93	1.081
A_1M	11.37	60	0.542	11.22	60	0.549	20.25	77	1.232
ML_Laplace	28.33	†	9.320	0.00	0	0.000	37.07	†	9.189
bcsstk16	0.32	78	0.139	0.33	71	0.126	0.38	74	0.131
raefsky2	0.39	117	0.202	0.39	117	0.200	0.51	119	0.204
lung2	0.50	70	0.226	–	–	–	0.71	24	0.257
FEM_3D_thermal2	2.91	11	0.060	2.85	10	0.055	3.81	12	0.048
Poisson3Db	11.63	143	0.484	2.95	136	0.620	7.17	†	3.693
thermal2	12.65	†	12.341	11.63	†	11.425	16.85	†	18.621
thermomech_TC	1.05	4	0.021	0.83	3	0.012	1.11	4	0.020
nlpkkt80	25.63	†	12.167	–	–	–	275.74	†	83.389
parabolic_fem	4.79	559	3.362	3.49	603	3.158	8.91	683	7.837
pde060	2.00	45	0.173	2.07	45	0.194	4.06	46	0.258
pde080	4.68	56	0.293	4.97	56	0.345	10.50	56	0.517
pde100	9.06	77	0.653	9.32	62	0.570	20.59	49	0.850

7. Conclusions

We have revisited some aspects of preconditioners based on approximations of the inverse of sparse matrices. We reviewed some algorithms based on the inversion and sparsification of incomplete factors and on inexact biconjugation. Some new results on their computational construction costs show that the biconjugation and the threshold-based sparse inversion have a setup cost that is quite similar.

The observations in Section 4 have been confirmed by the numerical experiments, where we made use of the PSBLAS/MLD2P4 framework. Setup of the preconditioners has been performed on a CPU platform, while the solution phase has been carried out on an NVIDIA GPU accelerator. The *AINV* preconditioner setup cost has been measured to be within a modest factor of an equivalent *INVT* preconditioner, and sometimes even better.

Carrying out the solution phase on the GPU platform has been demonstrated to be effective in terms of performance, making the approximate inverse preconditioners appealing on such a platform, as opposed to the situation on conventional CPUs where incomplete factorization preconditioners employing sparse triangular systems are usually more effective. This is true even if approximate inverses usually suffer a disadvantage in terms of number of iterations to convergence. The speedups that can be obtained are in line with the expectations from the evaluation of the computational kernels of sparse matrix–vector products. Very small test cases may not be amenable to an effective parallelization on GPUs because of their architectural features. In particular, they need a very large number of threads, each with its own workload, to achieve full exploitation of the computational capabilities.

Further work will be needed for an implementation able to take advantages of parallelism potentialities in INVT/INVK construction phase. In particular, we would like to clarify issues such as the effects of renumbering, scaling strategies, and estimation of preconditioner parameters. We also plan to embed the considered preconditioners in the updating framework

proposed in [5,8] and as local solvers in parallel Schwarz and algebraic multigrid-type frameworks. Finally, we plan to set up a package based on PSBLAS/MLD2P4 framework for approximate inverse software freely available in the near future.

Acknowledgments

We wish to thank two anonymous referees for comments that have improved this presentation. Moreover, thanks also to Dr. Pasqua D'Ambra of CNR Italy for her help in running the test cases. This work was supported in part by CINECA under the ISCRA grant programme for 2014, project IsC14_HyPSBLAS, and by Amazon with the AWS in Education Grant programme 2014.

References

[1] M. Benzi, Preconditioning techniques for large linear systems: A survey, J. Comput. Phys. 182 (2002) 418–477.
[2] D. Barbieri, V. Cardellini, S. Filippone, Generalized GEMM applications on GPGPUs: Experiments and applications, in: Proc. of 2009 Int'l Conf. on Parallel Computing, ParCo 2009, IOS Press, 2009.
[3] V. Cardellini, S. Filippone, Damian Rouson, Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms, Sci. Program. 22 (2014) 1–19.
[4] J.W. Choi, A. Singh, Richard W. Vuduc, Model-driven autotuning of sparse matrix–vector multiply on GPUs, SIGPLAN Not. 45 (2010) 115–126.
[5] M. Benzi, D. Bertaccini, Approximate inverse preconditioning for shifted linear systems, BIT 43 (2003) 231–244.
[6] D. Bertaccini, Efficient preconditioning for sequences of parametric complex symmetric linear systems, Electron. Trans. Numer. Anal. 18 (2004) 49–64.
[7] D. Bertaccini, F. Sgallari, Updating preconditioners for nonlinear deblurring and denoising image restoration, Appl. Numer. Math. 60 (2010) 994–1006.
[8] S. Bellavia, D. Bertaccini, B. Morini, Nonsymmetric preconditioner updates in Newton–Krylov methods for nonlinear systems, SIAM J. Sci. Comput. 33 (2011) 2595–2619.
[9] M.M. Dehnavi, D.M. Fernandez, J.L. Gaudiot, D.D. Giannacopoulos, Parallel sparse approximate inverse preconditioning on graphic processing units, IEEE Trans. Parallel Distrib. Syst. 24 (2013) 1852–1862.
[10] A.C.N. van Duin, Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices, SIAM J. Matrix Anal. Appl. 20 (1999) 987–1006.
[11] M. Benzi, D.B. Szyld, A. van Duin, Orderings for incomplete factorization preconditioning of nonsymmetric problems, SIAM J. Sci. Comput. 20 (1999) 1652–1670.
[12] M. Benzi, M. Tůma, Orderings for factorized sparse approximate inverse preconditioners, SIAM J. Sci. Comput. 21 (2000) 1851–1868.
[13] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., SIAM, Philadelphia, PA, 2003.
[14] A.V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Data Structures and Algorithms, Addison-Wesley, 1983.
[15] M. Benzi, M. Tůma, A sparse approximate inverse preconditioner for nonsymmetric linear systems, SIAM J. Sci. Comput. 19 (1998) 968–994.
[16] M. Benzi, J.K. Cullum, M. Tůma, Robust approximate inverse preconditioning for the conjugate gradient method, SIAM J. Sci. Comput. 22 (2000) 1318–1332.
[17] M. Benzi, C.D. Meyer, M. Tůma, A sparse approximate inverse preconditioner for the conjugate gradient method, SIAM J. Sci. Comput. 17 (1996) 1135–1149.
[18] A. Rafiei, F. Toutounian, New breakdown-free variant of ainv method for nonsymmetric positive definite matrices, J. Comput. Appl. Math. 219 (2008) 72–80.
[19] I.S. Duff, G.A. Meurant, The effect of ordering on preconditioned conjugate gradients, BIT 29 (1989) 635–657.
[20] R. Bridson, W.-P. Tang, Ordering, anisotropy and factored sparse approximate inverses, SIAM J. Sci. Comput. 21 (1999) 867–882.
[21] S. Demko, W.F. Moss, P.W. Smith, Decay rates for inverses of band matrices, Math. Comp. 43 (1984) 491–499.
[22] S. Filippone, M. Colajanni, PSBLAS: a library for parallel linear algebra computations on sparse matrices, ACM Trans. Math. Software 26 (2000) 527–550.
[23] S. Filippone, A. Buttari, Object-oriented techniques for sparse matrix computations in Fortran 2003, ACM Trans. Math. Software 38 (2012).
[24] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[25] D. Barbieri, V. Cardellini, S. Filippone, Damian Rouson, Design patterns for scientific computations on sparse matrices, in: Proc. of HPSS 2011, Springer, 2011.
[26] P. D'Ambra, D. di Serafino, S. Filippone, MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95, ACM Trans. Math. Software 37 (2010).
[27] D. Bertaccini, S. Filippone, Approximate inverse preconditioners for krylov methods on heterogeneous parallel computers, in: M. Bader, A. Bode, H. Bungartz, M. Gerndt, G. Joubert, F. Peters (Eds.), Parallel Computing: Accelerating Computational Science and Engineering, IOS Press, 2014, pp. 183–192.
[28] N. Bell, M. Garland, Implementing sparse matrix–vector multiplication on throughput-oriented processors, in: Proc. of Int'l Conf. on High Performance Computing Networking, Storage and Analysis, SC '09, ACM, 2009.
[29] R. Farina, S. Cuomo, P. De Michele, M. Chinnici, Inverse preconditioning techniques on a GPUs architecture in global ocean models, in: N.E. Mastorakis, Z. Bojkovic (Eds.), Recent Researches in Applied Mathematics and Informatics, Montreaux, Switzerland, December 29–31, in: Proceedings of the 16th WSEAS International Conference on Applied Mathematics, WSEAS Press, 2012, pp. 15–20.
[30] M. Geveler, D. Ribbrock, D. Goddeke, P. Zajac, S. Turek, Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses, Comput. Fluids 80 (2013) 327–332.
[31] S. Xu, W. Xue, K. Wang, H.-X. Lin, Generating approximate inverse preconditioners for sparse matrices using CUDA and GPGPU, J. Algorithms Comput. Technol. 5 (2011) 475–500.
[32] M. Naumov, Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS, Tech. Report., NVIDIA Corporation, 2011.
[33] H. van der Vorst, Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution o nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 13 (1992) 631–644.
[34] D. Bertaccini, G.H. Golub, S. Serra-Capizzano, Spectral analysis of a preconditioned iterative method for the convection–diffusion equation, SIAM J. Matrix Anal. Appl. 29 (2007) 260–278.
[35] P. D'Ambra, S. Filippone, A parallel generalized relaxation method for high-performance image segmentation on GPUs, J. Comput. Appl. Math. (2015).
[36] S. Filippone, P. D'Ambra, M. Colajanni, Using a parallel library of sparse linear algebra in a fluid dynamics applications code on linux clusters, in: G. Joubert, A. Murli, F. Peters, M. Vanneschi (Eds.), Parallel Computing — Advances & Current Issues, Imperial College Press, 2002, pp. 441–448.
[37] G. Bella, F. Bozza, A. De Maio, F. Del Citto, S. Filippone, An enhanced parallel version of KIVA-3V coupled with a 1D CFD code and its use in general purpose engine application, in: M. Gerndt, D. Kranzlmuller (Eds.), Proceedings of HPCC 2006, in: LNCS, vol. 4208, Springer Verlag, 2006, pp. 11–20.
[38] T.A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Software 38 (2011) 1:1–1:25.
[39] M. Benzi, M. Tůma, A comparative study of sparse approximate inverse preconditioners, Appl. Numer. Math. 30 (1999) 305–340.
[40] P.R. Amestoy, T.A. Davis, I.S. Duff, Algorithm 837: AMD, an approximate minimum degree ordering algorithm, ACM Trans. Math. Software 30 (2004) 381–388.
[41] P. Amestoy, T.A. Davis, I.S. Duff, An approximate minimum degree ordering algorithm, SIAM J. Matrix Anal. Appl. 17 (1996) 896–905.
[42] N.E. Gibbs Jr., W.G. Poole, P.K. Stockmeyer, A comparison of several bandwidth and profile reduction algorithms, ACM Trans. Math. Software 2 (1976) 322–330.
[43] N.E. Gibbs Jr., W.G. Poole, P.K. Stockmeyer, An algorithm for reducing the bandwidth and profile of a sparse matrix, SIAM J. Numer. Anal. 18 (1976) 235–251.