PIVOTING STRATEGY FOR FAST LU DECOMPOSITION OF SPARSE BLOCK MATRICES

Lukas Polok
Brno University of Technology,
Faculty of Information Technology,
IT4Innovations Centre of Excellence
Bozetechova 1/2, Brno 61266, Czech Republic
ipolok@fit.vutbr.cz

Pavel Smrz
Brno University of Technology,
Faculty of Information Technology,
IT4Innovations Centre of Excellence
Bozetechova 1/2, Brno 61266, Czech Republic
smrz@fit.vutbr.cz

ABSTRACT

Solving large linear systems is a fundamental task in many interesting problems, including finite element methods (FEM) or (non-)linear least squares (NLS) for inference in graphical models such as simultaneous localization and mapping (SLAM) in robotics or bundle adjustment (BA) in computer vision. Furthermore, the problems of interest here are sparse. The most time-consuming parts are sparse matrix assembly and linear solving. An interesting property of these problems is their block structure. The variables exist in multi-dimensional space such as 2D, 3D or $\mathfrak{se}(3)$ and hence their respective derivatives are dense blocks. In our previous work (Polok et al. 2013), we demonstrated the benefits of explicitly representing blocks in sparse matrices, namely faster matrix assembly and arithmetic operations. Here, we propose and evaluate a novel sparse block LU decomposition. Our algorithm is on average $3\times$ faster (best case $50\times$), causes less fill-in and has comparable or often better precision than the conventional methods.

Keywords: LU decomposition, sparse matrix, block matrix, register blocking, direct methods.

1 INTRODUCTION

Solving a linear system of the form $Ax = b$ is not trivial, unless A has specific properties such as being a triangular matrix. Then, the last element of the unknown vector x is given by a ratio of the corresponding elements in A and b. Next, the second last element can be calculated by substituting the first element and solving another simple ratio and so on, until all of x is recovered–a process called *back-substitution*. There are several algorithms for bringing a matrix into triangular form, Gaussian elimination, LU decomposition, Cholesky factorization and QR decomposition (Davis 2006). These algorithms decompose the original matrix A into a product of two or more factor matrices that are triangular or otherwise easily invertible (diagonal or orthogonal). The solution to the original linear system then becomes a sequence of solutions for each of those factors, ultimately yielding x. In the case of LU decomposition, this amounts to first solving $Ly = b$ to get an intermediate vector y, and then solving $Ux = y$.

Notably, Cholesky factorization has the lowest time complexity of all these algorithms. It decomposes $A = R^\top R$ where R is upper-triangular matrix identical to that in QR decomposition, up to the sign (Cholesky will always yield R with positive diagonal entries). It is required that A be square, positive-definite (SPD) matrix. Matrices involved in least squares problems are SPD, which makes Cholesky an attractive choice for this class of applications. The disadvantage of using Cholesky for over-determined systems is the need to form a part of the Moore-Penrose pseudoinverse, $A^+ = (A^\top A)^{-1}A^\top$ that typically results in solving a system in the form $(A^\top A)x = A^\top b$. The formation of the square matrix $A^\top A$ may in some cases increase the amount of memory needed and does increase the condition number, making the system more difficult to solve from numerical precision point of view.

If the matrix A is sparse, there is another interesting facet to the Cholesky decomposition, the *fill-in*. Ideally, R would have the same sparsity pattern as the upper triangle of A. However, in the course of calculating the factorization, other non-zero entries may be introduced in R and those are referred to as the fill-in. The number of nonzeros in R directly affects the speed of the factorization and subsequent backsubstitution. The amount of fill-in is dependent on the ordering of matrix rows and columns and fill-reducing orderings have been proposed in the literature, most notably the minimum degree ordering (George and Liu 1989, Liu 1985) and the faster approximate minimum degree (AMD) ordering (Amestoy, Davis, and Duff 1996).

The older LU decomposition is more general than Cholesky, and can factorize any square, invertible matrix into a product $A = LU$, where L is a lower-triangular matrix with unit diagonal and U is a general upper triangular matrix. This means the matrix no longer needs to be SPD. Since the diagonal of L is always the same, it does not need to be stored and sometimes L and U are stored together in a single matrix. Similar to Cholesky, LU decomposition also introduces fill-in and it is also possible to use the AMD algorithm for finding a fill-reducing ordering.

Unlike Cholesky factorization, LU is not inherently numerically stable and requires *pivoting*. A pivot is an element of the diagonal of A, that will serve as a divisor for other values in the decomposition algorithm. The magnitude of the pivot is of great importance if the numerical precision is finite. Using a small pivot will amplify the values in the matrix and lead to roundoff errors. The pivoting schemes therefore choose the largest pivot, either from the current column (referred to as *partial* pivoting) or from the so far unreduced remainder of the matrix, *full* pivoting.

While these strategies improve numerical stability, they also cause row or row and column reordering. This subsequently interferes with the fill-reducing ordering of the matrix and may inadvertently increase fill-in to unacceptable levels. In this paper, we show that block-based pivoting helps to reduce this effect, while at the same time not destroying the precision of the result. This strategy is a proof of concept and shows that block pivoting is possible and applicable to all kinds of decompositions that require it, be it LU, QR or LDL$^\top$ (a different form of Cholesky factorization that can work with symmetric indefinite matrices but which requires pivoting). We chose the LU factorization to demonstrate it because it is relatively simple to implement and has a potentially greater practical impact than LDL$^\top$.

The remainder of this paper is structured as follows. The next section summarizes related work, especially from the point of view of related sparse block matrix research and of pivoting strategies that were proposed in the literature. Section 3 briefly revisits principles of LU decomposition and common forms of algorithms for performing it. Section 4 describes the design of the proposed algorithm and how it differs from the standard LU decomposition methods implemented in packages such as CSparse (Davis 2006). Finally, Section 5 describes the evaluation of the proposed method.

2 RELATED WORK

While the formats for representing sparse block matrices date back to the early basic linear algebra subprograms (BLAS) proposal (Du and Marrone 1992), there are surprisingly only a few implementations that support them. The most popular sparse matrix package, CSparse (Davis 2006), a part of SuiteSparse only supports element-wise sparse matrices in compressed sparse column (CSC) format. Despite that, it is also being extensively used in applications where block matrices occur.

E.g. Google's Ceres solver (Agarwal and Mierle 2012), a NLS implementation behind popular products such as 3D maps or Street View uses CSparse for linear solving (other choices are also available, though, by using a dense solver or an iterative one). It also defines its own block storage format that serves to accelerate the sparse matrix assembly, but this format is abandoned in favor of CSC as soon as arithmetic operations on the matrix are required.

NIST Sparse BLAS (Carney, Heroux, Li, and Wu 1994) supports matrices in the compressed sparse row (CSR) format and in addition also the block compressed sparse row (BSR), a format for block matrices where all the blocks in a single matrix are the same size, and variable block compressed sparse row (VBR), a general format for block matrices where a single matrix can contain blocks of different dimensions. Unfortunately, there are no algorithms for matrix decompositions in NIST Sparse BLAS, nor is there an associated package that would contain them. Also, the triangular solving options are limited–only matrices with unit diagonal are supported.

There are more libraries that support the BSR format, most notably Intel MKL (F. 2009) or PETsc (Balay et al. 2015). Those can be readily used for solving linear systems, although limited to matrices containing only square blocks of a single dimension. This effectively limits their use to simpler problems with only variables of a single type (multiple variable types would in most cases yield blocks of several different dimensions and thus also rectangular blocks).

In our previous work (Polok, Ila, and Smrž 2013), we proposed an efficient block matrix storage format and several algorithms for performing arithmetic operations. The results were compared to CSparse and Ceres, proving the proposed implementation superior. We furthermore demonstrated the ability to outperform other block matrix implementations used in robotics (Polok, Šolony, Ila, Zemčík, and Smrž 2013). Later on, an implementation of sparse block Cholesky was added and its variant for incremental solving was also proposed (Polok, Ila, Šolony, Smrž, and Zemčík 2013).

Blocking is a popular technique for attaining higher memory throughput in dense implementations, used e.g. in LAPACK (Anderson et al. 1987). In Eigen (Guennebaud, Jacob, et al. 2010), the partially pivoted LU decomposition is blocked, splitting the matrix to rectangular blocks. Each such block contains a part of the diagonal and all the elements under it, so that the blocking would not interfere with pivot choice. After decomposing this block, the changes are communicated to the lower-right submatrix in a blockwise manner.

Ultimately, the choice of pivoting algorithm can have a great impact on the performance, due to required communication and memory access patterns. There were many pivoting algorithms proposed in the literature. MA21 (Duff 1981a, Duff 1981b) is one of the early examples, producing such a row permutation that the matrix ends up having nonzero diagonal entries. While not the best pivoting strategy for sparse decompositions, it showed potential in improving iterative solver convergence. The latter work (Duff and Koster 1999) expanded into obtaining such an ordering that the magnitude of the diagonal entries is maximized. It explores maximum product of the diagonal entries, which is the pivoting strategy applied in this paper.

Parallel implementations of LU decomposition often try to avoid pivoting during the decomposition phase itself, often by using threshold pivoting (a pivot permutation only takes place if its magnitude is larger by a given threshold than that of the natural pivot), or by performing static pivoting beforehand (Li and Demmel 2003). This helps to reduce communication and synchronization otherwise required.

Schenk and Gärtner (2006) propose a pivoting strategy for the $LDL^\top$ factorization, not entirely unlike the method proposed here. Their algorithm chooses pivots of size 1×1 or 2×2 that are factorized in blockwise fashion (and in the case of 2×2, the blocks themselves are also subject to intra-block pivoting that the authors refer to as *perturbation*). In this work, we use pivoting at the granularity of the naturally occurring blocks, rather than choosing the size of the pivots.

Algorithm 1 Two Dense LU Decomposition Algorithms.

Require: That A is an invertible $n \times n$ matrix, P is $n \times n$ identity matrix.

```
1: function PIVOTING($A$, $P$, $k$)
2:     $p = \text{CHOOSEPIVOT}(A_{k:end,k})$        ▷ Choose a pivot from the lower portion of column $k$.
3:     if $p \neq k$ then
4:         SWAP($A_{k,*}$, $A_{p,*}$)        ▷ Swap the pivotal row with the next unreduced row in $A$.
5:         SWAP($P_{k,*}$, $P_{p,*}$)        ▷ Swap the same rows in the permutation matrix $P$.
6:     end if
7:     return $(A, P)$
8: end function

9: function SUBMATRIXLU($\Lambda$)
10:     for $k = 0$ to $n-1$ do        ▷ For each column in $A$.
11:         ($A$, $P$) = PIVOTING($A$, $P$, $k$)
12:         for $i = k+1$ to $n-1$ do
13:             $A_{i,k} = A_{i,k} / A_{k,k}$        ▷ Divide by the chosen pivot.
14:         end for
15:         for $j = k+1$ to $n-1$ do        ▷ Right-looking, exclusive.
16:             for $l = k+1$ to $n-1$ do
17:                 $A_{l,j} = A_{l,j} - A_{l,k} \cdot A_{k,j}$        ▷ Scatter contributions to the so far unreduced submatrix.
18:             end for
19:         end for
20:     end for
21: end function

22: function COLUMNLU($\Lambda$)
23:     for $k = 0$ to $n-1$ do        ▷ For each column in $A$.
24:         for $j = 0$ to $k-1$ do        ▷ For all elements strictly above the pivot.
25:             $A_{j,k} = A_{j,k} / A_{j,j}$        ▷ Divide U by the past pivots.
26:             for $l = j+1$ to $n-1$ do        ▷ Left-looking.
27:                 $A_{l,k} = A_{l,k} - A_{j,k} \cdot A_{l,j}$        ▷ Gather contributions from the already factorized columns.
28:             end for
29:         end for
30:         ($A$, $P$) = PIVOTING($A$, $P$, $k$)
31:         for $i = k+1$ to $n-1$ do
32:             $A_{i,k} = A_{i,k} / A_{k,k}$        ▷ Divide L by the chosen pivot.
33:         end for
34:     end for
35: end function
```

3 LU DECOMPOSITION

In this section, the basic algorithm for LU decomposition is revised, to give insights how the blocked algorithm will be implemented. Two basic transformation of a dense algorithm are in Algorithm 1. The SUBMATRIXLU is right-looking version of the algorithm and it is a common way of implementing dense LU decomposition. It is right-looking and produces one column of L and one row of U at a time. This is sometimes referred to by the order of the loops, as the `kij` algorithm.

The (partial) pivoting is performed by choosing a particular element from the lower part of the current column of A (Algorithm 1, line 2). The choice is typically performed as:

$$p_k = \underset{j}{\operatorname{argmax}} \left(|A_{j,k}| \cdot w_j \cdot \begin{cases} 1+t & \textit{if } j = k \\ 1 & \textit{otherwise} \end{cases} \right), \; j \geq k \;, \tag{1}$$

where w_j is approximate pivot weight vector, determined by taking row-wise L_∞ norm of A or 1 if no weighting is used and t is pivot threshold or 0 if no pivot thresholding is used. Upon choosing a pivot, the corresponding row is swapped with the current row k and this change is collected in the permutation matrix P (lines 4 and 5). If the weight vector w was used, the same swap would be performed there as well. To perform full pivoting, one would choose a pivot from the entire submatrix $A_{i,j} \mid i \geq k \wedge j \geq k$.

This algorithm yields a decomposition $LU = PA$, where the matrices overwrite A with $LU - I$, where I is an identity matrix–the unit diagonal of L that is not explicitly stored. This is a common way of representing the decomposition in the dense case.

The same algorithm is, however, not well suited for direct implementation of a *sparse* decomposition, as it requires access to both rows and columns of the matrix. If using a sparse storage format such as CSC, the matrix access pattern needs to be by columns–accessing the matrix by rows amounts to searching for every element and would be overly costly. Instead, by changing the order of the loops to `kji`, it is possible to arrive at COLUMNLU that only requires column-wise access. It is a left-looking algorithm and produces one column of the factorization at a time.

Conceptually, the first half of this algorithm is triangular solving (lines 24 to 29) and the rest is just choosing the pivot and column scaling. Note that in sparse case, swapping rows for pivoting would be inefficient and the implementations instead maintain a row permutation and simply renumber rows of all elements at the end (Davis 2006). Also note that due to always having only a single unreduced column, full pivoting is not easily attained. In the sparse case, the L and U matrices are stored separately, as it saves the work of searching for the diagonal elements when back-substituting later on.

4 PROPOSED ALGORITHM

The proposed algorithm is based on the procedure COLUMNLU described in the previous section, with several differences. The key difference is the use of a sparse *block* structure described in (Polok, Ila, and Smrž 2013). It is a column-major data structure similar to VBR (Du and Marrone 1992). It allows blocks of different sizes in a single matrix where the edges of the blocks must be aligned with each other, forming non-overlapping block rows and block columns. Each block is stored as a dense matrix, with the elements of all blocks being serialized in a single pooled array. This improves cache coherency of in-order traversal of the elements. Care is also taken for the blocks to be aligned in memory to allow vectorization using SSE instructions.

From the algorithmic point of view, each $A_{i,j}$ is a dense matrix rather than a scalar value. Thus, the product at line 27 of Algorithm 1 is actually a dense matrix product. Similarly, the division at line 25 is triangular solution of the form $L^{-1}_{j,j} \cdot U_{j,k}$ and the division at line 32 is another triangular solution, this time of the form $L_{i,k} \cdot U^{-1}_{k,k}$ where the triangular block $U_{k,k}$ is on the *right*. Those expressions are both evaluated using forward and back-substitution, respectively.

Another difference is the choice of the pivot block, which we refer to as the inter-block pivoting. For element-wise sparse matrices, this can be done according to (1). For blockwise matrices, this formula cannot be used directly and a way of reducing the blocks to scalar values needs to be devised. We tested a number of different metrics, including trace or L_1, L_2 and L_∞ norms of the block or of its diagonal, to no avail. Finally, using a product of diagonal entries in a block permuted to have the largest diagonal values (Duff and Koster 1999) gave reasonable results. This stems from the fact that all the off-diagonal elements will be in sequence divided by all the diagonal elements of the pivot block in the back-substitution mentioned above, and thus the final scaling is equal to their product. We also use pivot weighting by taking block row-wise L_∞ norm of A. This helps to choose better pivots in matrices with uneven distribution of off-diagonal value magnitudes.

Packages such as CSparse perform pivoting by element renumbering, leading to unsorted CSC matrices (the order of elements in each column is undefined). This potentially creates suboptimal memory access patterns if the number of elements in each column is greater than the effective size of the CPU cache. The proposed implementation addresses this problem differently and produces and maintains an ordered representation at all times. To do that, a method described in (Gustavson 1978) is employed: a dense vector of the same size as the current block column is used, along with a bit array to accumulate the values of the blocks and the sparsity pattern, respectively. Once the decomposition of the current block column is finished, the contents of this dense accumulator are scattered to the L and U matrices, in order.

An important difference is the factorization of the pivot block. This is a simple dense LU factorization and brings a choice of partial or full intra-block pivoting (Interestingly, the factorization of the pivot block does not immediately depend on or affect any other blocks.). Full pivoting has the advantage of revealing the rank of this pivot block. If the block is rank deficient, care must be taken when evaluating $L_{i,k} \cdot U_{k,k}^{-1}$. If the columns of $L_{i,k}$ corresponding to the zero diagonal entries in $U_{k,k}$ are also null, the division needs to be avoided otherwise the floating-point arithmetics would produce special values. If, on the other hand, those columns are nonzero, using this pivot would lead to division by zero and a different pivot needs to be chosen. If there is no full-rank pivot block in the current block column then either the factorization needs to fail, or elements from multiple different blocks would need to be combined. In the proposed implementation, this problem was handled by failing the factorization since it did not occur in the tested matrices.

In any case, the intra-block pivoting required by the pivot block factorization potentially reorders the rows and columns of this block and the changes need to be reflected on the rest of the matrix. While pivoting the columns affects only the current block column and can be applied immediately to all the other blocks, pivoting the rows affects the already processed block row of L and future block rows in U. Since accessing the matrix row-wise is prohibitive, the row permutation is only applied to the rows of U as they are produced and the permutation in L is performed at the end, after the factorization finishes.

5 EXPERIMENTAL EVALUATION

The proposed algorithm was evaluated on matrices from the University of Florida Sparse Matrix Collection (Davis 1994). Since the matrices in this dataset do not contain any information about block structure, a modified algorithm, based on routines `CSRKVSTR` and `CSRKVSTC` described in Saad (1994), was used to find block matrices with a particular block size and allowing a small amount of fill-in. In addition to that, we compared the implementations on block matrices associated with some standard SLAM problems in robotics: *Parking Garage* (Kümmerle et al. 2011), *KITTI Sequence 00* (Geiger et al. 2013), *Sphere 2500* (Kaess, Ranganathan, and Dellaert 2007) and BA problems in computer vision: *Fountain-P11* (Strecha et al. 2008), *Lourakis bundle1* (Lourakis and Argyros 2004), *Mazaheri bundle_-adj* (Davis 1994), *Venice871* (Kümmerle et al. 2011), *Fast & Furious 6* (Double Negative Visual Effects, http://www.dneg.com/.) and *Guildford Cathedral* (http://cvssp.org/impart/.).

All the matrices were pre-ordered using the same fill-reducing ordering, obtained by AMD of $AA^\top$ with dense columns dropped and applied at the granularity of blocks. This means that all the tested methods operated on identical inputs. The time to produce this ordering is not included in the timing results (since all the methods would use the same ordering scheme). Ultimately, this slightly favors element-wise approaches since the time complexity of the ordering algorithm is higher than linear in the size of the matrix (Heggernes, Eisestat, Kumfert, and Pothen 2001) and thus ordering at the level of blocks is faster than ordering at the level of elements would be.

The experiments were performed on the Salomon supercomputer, part of the IT4I Czech National Supercomputing Center. Each compute node is equipped with a pair of 12-core Xeon E5-2680 v3 running at 2.50 GHz and 128 GB of RAM. Note that these CPUs have turbo boost technology which adjusts the clock frequency based on the available thermal envelope. This function was disabled for the benchmarks, so as to not make the results dependent on the variations in the temperature. All of the processing times would be lower with turbo boost enabled. The code was compiled as x64, and used 64-bit pointers. During the tests, the computer was not running any time-consuming processes in the background. We used the `g++` (`gcc`) 4.4.7 compiler (the proposed implementation is written in C++, while CSparse is written in C).

Each test was run at least ten times and until cumulative time of at least 5 seconds was reached, and the average time was calculated in order to avoid measurement errors, especially on smaller matrices. Note that each timing run was performed in a new process, so that there would be no cache reuse. We further experimented with flushing the cache lines containing the data, using the combination of `_mm_clflush()` and `_mm_mfence()` intrinsics. Furthermore, a 100 MB block of memory was read and written to make sure that the cache was completely flushed (care was taken that these accesses would not bypass the cache).

The effect of thus flushed cache was a small slow-down, on average 3.80% for CSparse and 2.86% for the proposed method (worst case 25.86% for CSparse and 14.43% for proposed). This effect was more pronounced on smaller matrices, as the larger matrices do not fit in the cache at once anyway. The flushing of the cache did not change the ranking of the methods on any of the tested matrices. The tests presented in the remainder of the evaluations herein are without flushing the cache, as it seems more natural that the matrix to be factorized is already (partially) in the cache (since the factorization function would be most likely called on a matrix that was just produced by other computation). It also makes the presented results more comparable to the results of other researchers. But note that there is still no cache reuse between individual benchmark runs as those are performed each in a new process.

Apart from the obvious timing evaluation and also recording the statistics about the factorization sparsity, relative factorization precision was evaluated, as:

$$\frac{\|PAQ - LU\|}{\|A\|} , \tag{2}$$

where P and Q are the block row and column permutation matrices (Q is only used in the proposed implementation, if full intra-block pivoting is applied).

Results for the SLAM and BA datasets can be seen in Table 1. In these datasets, the block partitioning is easily anticipated (unlike in the rest of the benchmarks where the block structure was estimated and might not exactly map to the original variables in some cases). Note that the system matrices of these datasets are symmetric but neither implementation took advantage of this fact, and the numbers of nonzeros (denoted "Nnz.") are reported for both halves of the matrix. It is a common practice in solving BA problems to apply Schur complement as:

$$A = \begin{pmatrix} C & U \\ V & L \end{pmatrix} \tag{3}$$

$$\mathrm{Schur}(A) = C - UL^{-1}V , \tag{4}$$

where the variables in A are partitioned in such a way that C contains the camera variables and L contains the landmark variables. Due to the structure of the problem, L is block diagonal and easily invertible. The solution to the linear system $Ax = b$ then lies in decomposition of $\mathrm{Schur}(A)$ rather than of the entire A. To

Table 1: Results on SLAM and BA datasets. The times are in milliseconds unless specified otherwise. The first group are SLAM datasets, followed by a BA dataset and finally Schur-complemented BA datasets.

			CSparse			**Proposed**		
Name	**Size**	**Nnz.**	**Time**	**LU nnz.**	**Error**	**Time**	**LU nnz.**	**Error**
Garage	9966	285696	104.273	1135362	$8.16\cdot10^{-16}$	69.415	936360	$3.53\cdot10^{-16}$
KITTI 00	27246	477072	194.675	2078526	$6.48\cdot10^{-16}$	114.596	1673244	$1.90\cdot10^{-16}$
Sphere 2500	15000	268164	2597.737	6557052	$1.46\cdot10^{-15}$	1250.732	5190048	$1.04\cdot10^{-15}$
Fountain	23853	554427	1921.963	10602408	$3.49\cdot10^{-15}$	68.678	1112814	$1.73\cdot10^{-15}$
Lourakis	3115	31572	20.703	88473	$7.55\cdot10^{-16}$	13.011	86436	$2.75\cdot10^{-15}$
Mazaheri	3330	247068	149.431	793026	$5.51\cdot10^{-16}$	80.813	614880	$2.94\cdot10^{-16}$
Venice871	5226	5469048	125.990 s	26586570	$4.30\cdot10^{-15}$	57.254 s	26439624	$1.17\cdot10^{-15}$
FF6	960	137160	291.344	702228	$1.21\cdot10^{-15}$	162.147	604584	$2.30\cdot10^{-16}$
Cathedral	552	125244	82.581	278310	$1.86\cdot10^{-15}$	54.886	263592	$3.06\cdot10^{-16}$

reflect this in the tests performed here, the matrices of BA datasets in Table 1 are first Schur-complemented and then the results are reported for the LU decomposition of this Schur complement. An exception was made for the *Fountain* dataset due to its small size–the resulting times would be very close to zero.

The proposed implementation gets consistently better times and better precision, with the exception of the *Lourakis* dataset. The precision is in the 10^{-16} to 10^{-15} range, which corresponds to the roughly 15 digits that the double precision floating-point numbers can hold. The good speed is caused by the fact that these matrices are diagonally dominant and the proposed implementation can perform most of the pivoting inside of the blocks, reaching lower fill-in and thus also lower number of arithmetic operations than CSparse.

The results on the other matrices, from the University of Florida Sparse Matrix Collection, are in Table 2. The matrices are grouped by block size, starting with 4×4 and ending with 6×6. Although the collection contains much more block matrices, they typically contain mixtures of multiple block sizes. To limit the size of the evaluation to a reasonable size, we decided to only select matrices with a single block size (note that the implementation supports multiple block sizes, e.g. *Fountain-P11* contains 6×6, 6×3, 3×6 and 3×3 blocks).

While the good precision and sparsity holds up, the speedup grows with block size and the proposed implementation is slightly slower for 4×4 blocks. Such behavior is to be expected from blocked implementation where there are more nested loops and thus a larger ratio of control flow to arithmetics instructions. This could be easily improved by loop unrolling, e.g. as suggested in Polok, Ila, and Smrž (2013). Note that CSparse failed to factorize 12 of the tested matrices and so they were omitted to save space (on those matrices, the average relative error of the proposed implementation was $5.648\cdot10^{-16}$, with the worst case relative error being $1.363\cdot10^{-15}$). Additionally, several more matrices were omitted from groups of matrices having the same structure and getting the same results (e.g. *Schenk/AFE_af_shell1* through *Schenk/AFE_af_shell9* or the *Simon/venkat* group).

While already giving good results, there are several ways to improve the implementation to be even faster. In our previous work, we proposed fixed block size (FBS) optimization (Polok, Ila, and Smrž 2013), a form of register unrolling that is conveniently accessible by using the C++ language, making different block sizes or even their mixtures easily attainable via a simple interface and without having to manually rewrite or optimize any code. Using this optimization makes this method faster also on matrices with 3×3 blocks, the proposed method is faster than CSparse on 31 out of 33 matrices with average speedup $2.46\times$. The results of this optimization are plotted in Figure 1. This plot shows the speedup of the optimized algorithm compared to the results reported in Tables 1 and 2 as "Proposed". The speedup is greater for larger matrices and for smaller block sizes, especially for 3×3 and 4×4 that fit well in the SSE registers. The smaller gains

Table 2: Results on matrices from the University of Florida Sparse Matrix Collection. The times are in milliseconds unless specified otherwise. The first group are matrices with 4×4 blocks, followed by 5×5 and 6×6 blocks. Note that the names of the matrices were abbreviated in some cases, in order to save space.

			CSparse			**Proposed**		
Name	**Size**	**Nnz.**	**Time**	**LU nnz.**	**Error**	**Time**	**LU nnz.**	**Error**
HB/steam3	80	928	0.043	1068	$1.37 \cdot 10^{-17}$	0.132	1248	$5.36 \cdot 10^{-20}$
Simon/raefsky3	21200	1488768	8211.861	$149 \cdot 10^5$	$1.00 \cdot 10^{-15}$	8800.133	$133 \cdot 10^5$	$5.56 \cdot 10^{-16}$
Simon/venkat01	62424	1717792	5010.632	$178 \cdot 10^5$	$3.55 \cdot 10^{-16}$	6870.423	$179 \cdot 10^5$	$1.77 \cdot 10^{-16}$
Janna/CoupC3D	416800	$223 \cdot 10^5$	4.023 h	$151 \cdot 10^7$	$4.70 \cdot 10^{-16}$	3.103 h	$148 \cdot 10^7$	$5.25 \cdot 10^{-16}$
Oberwolf./piston	2025	100015	13.396	177445	$2.26 \cdot 10^{-16}$	14.486	148600	$6.91 \cdot 10^{-17}$
Fluorem/GT01R	7980	430909	1608.026	4446585	$7.95 \cdot 10^{-16}$	1232.581	3744400	$8.94 \cdot 10^{-05}$
Schenk/shell1	504855	$176 \cdot 10^5$	446.545 s	$382 \cdot 10^6$	$7.49 \cdot 10^{-15}$	356.576 s	$384 \cdot 10^6$	$1.32 \cdot 10^{-15}$
Schenk/shell10	$151 \cdot 10^4$	$527 \cdot 10^6$	2.541 h	$168 \cdot 10^7$	$5.07 \cdot 10^{-14}$	1.515 h	$168 \cdot 10^7$	$8.29 \cdot 10^{-15}$
Schenk/0_k101	503625	$176 \cdot 10^5$	486.978 s	$398 \cdot 10^6$	$6.74 \cdot 10^{-15}$	379.897 s	$399 \cdot 10^6$	$2.24 \cdot 10^{-15}$
HB/bcsstk02	66	4356	0.285	4422	$2.21 \cdot 10^{-16}$	0.394	4752	$1.49 \cdot 10^{-16}$
HB/bcsstk14	1806	63454	52.663	366174	$9.70 \cdot 10^{-16}$	28.607	281952	$3.20 \cdot 10^{-16}$
Nasa/nasasrb	54870	2677324	20.933 s	$407 \cdot 10^5$	$1.28 \cdot 10^{-15}$	8.769 s	$302 \cdot 10^5$	$6.15 \cdot 10^{-16}$
Simon/olafu	16146	1015156	3094.454	7964682	$4.25 \cdot 10^{-16}$	1739.891	6871176	$1.11 \cdot 10^{-13}$
DNVS/shipsec1	140874	7813404	1458.87 s	$314 \cdot 10^6$	$5.75 \cdot 10^{-16}$	295.690 s	$210 \cdot 10^6$	$9.31 \cdot 10^{-16}$
DNVS/x104	108384	$102 \cdot 10^5$	367.990 s	$174 \cdot 10^6$	$2.18 \cdot 10^{-15}$	73.593 s	$100 \cdot 10^6$	$8.90 \cdot 10^{-16}$
BenElechi/B.E.1	245874	$132 \cdot 10^5$	172.599 s	$180 \cdot 10^6$	$1.83 \cdot 10^{-15}$	101.651 s	$180 \cdot 10^6$	$7.23 \cdot 10^{-16}$

on large block sizes is given by the fact that those are already quite efficient even without this optimization. Note that the optimized method is never slower, and also that the precision of the results is identical to that of the unoptimized version.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an implementation of intra-/inter-block pivoting scheme based on the maximum diagonal product scoring of the pivot blocks. It serves as a showcase that limiting pivot search to relatively small blocks can yield excellent precision while at the same time promoting locality of reference and reducing the number of nonzero elements in the resulting factorization, as shown by the experimental evaluation. The proposed method was demonstrated on LU decomposition but is applicable also on other decomposition types, such as QR or $LDL^\top$.

The evaluation presented here was in comparison with CSparse. While it is very popular, it is a simplical, serial code. It would be interesting to compare the proposed algorithm to also other, more advanced implementations such as superLU (Li et al. 1999), MUMPS (Amestoy, Duff, L'Excellent, and Koster 2001) or HSL MA50 (Duff and Reid 1996). This evaluation would be well beyond scope (and space) of this study and we shall report it in a follow-up paper. A quantitative comparison of performance is in Figure 2, the proposed implementation peaks at 5.714 GFLOP/s, average is 2.537 GFLOP/s (single core performance). (The figures were arrived at by calculating the number of FLOPs required for the factorization, using the functionality described at https://sf.net/p/slam-plus-plus/wiki/Counting%20FLOPS%20in%20Sparse%20Matrix%20Operations/, and dividing that by average runtime (including the symbolic factorization) reported in Tables 1 and 2. The *Fountain-P11* dataset was excluded as the proposed implementation yields substantially more sparse factorization and the resulting figure would be unrealistic (31 GFLOP/s).)

The LU decomposition is also amenable to parallelization, which is another interesting direction, especially with respect to GPU implementation. Finally, the block methods are orthogonal to multifrontal and supern-

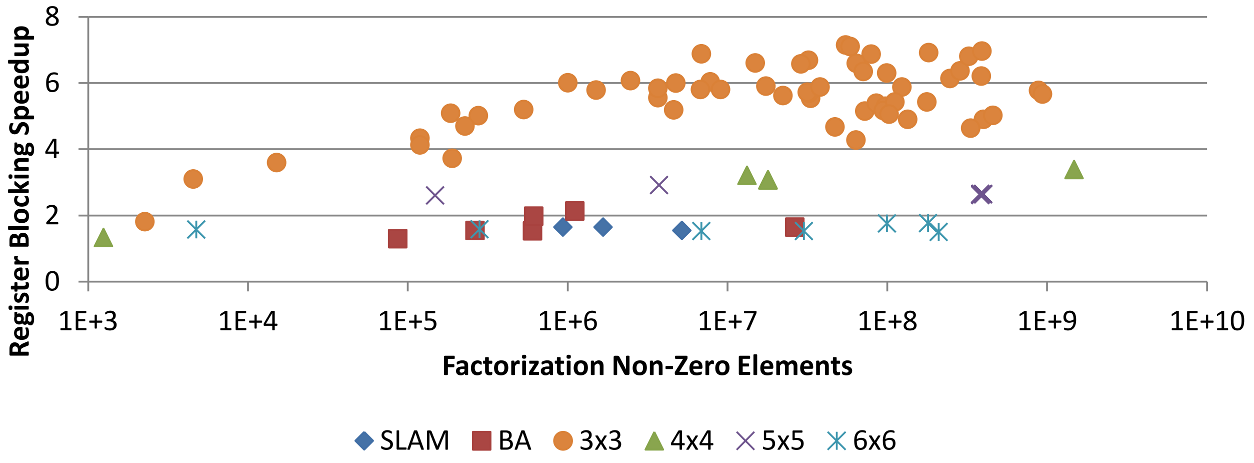

Figure 1: Relative speedup of the fixed block size (register blocking) optimization, compared to the unoptimized proposed algorithm.

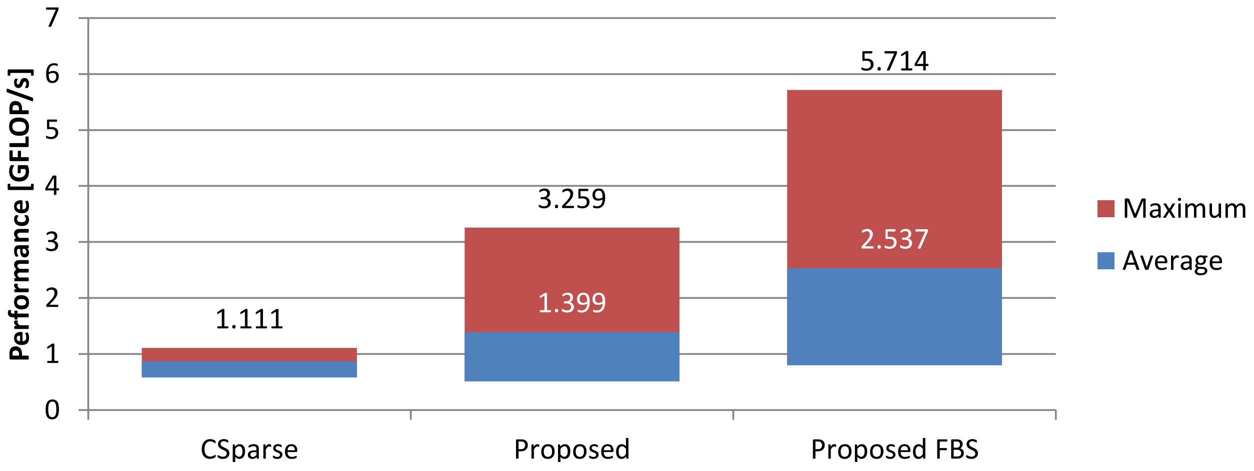

Figure 2: Quantitative evaluation of the compared algorithm performance in floating point operations per second (FLOP/s). The bottom of the bars indicates worst-case performance (0.56 GFLOP/s for CSparse, 0.50 GFLOP/s proposed and 0.78 GFLOP/s proposed with FBS optimization).

odal methods, that should both increase the performance even more, by using frontal matrices or supernode blocks and enabling dense computation on larger than the natural blocks.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science (LQ1602). The access to the Salomon supercomputer was supported by the IT4Innovations centre of excellence, via the Block LU Decomposition (BLUD) project (DD-17-5).

REFERENCES

S. Agarwal and K. Mierle 2012. "Ceres Solver". http://ceres-solver.org/.

Amestoy, P. R., T. A. Davis, and I. S. Duff. 1996. "An approximate minimum degree ordering algorithm". *SIAM J. on Matrix Analysis and Applications* vol. 17 (4), pp. 886–905.

Amestoy, P. R., I. S. Duff, J.-Y. L'Excellent, and J. Koster. 2001. "A fully asynchronous multifrontal solver using distributed dynamic scheduling". *SIAM J. on Matrix Analysis and Applications* vol. 23 (1), pp. 15–41.

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney et al. 1987. *LAPACK Users' guide*, Volume 9. SIAM.

Balay, S., S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. 2015. "PETSc Users Manual". Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory.

Carney, S., M. A. Heroux, G. Li, and K. Wu. 1994. "A Revised Proposal for a Sparse BLAS Toolkit". Technical report, SPARER.

Davis, T. 1994. "The University of Florida Sparse Matrix Collection". In *NA Digest*. Citeseer.

Davis, T. A. 2006. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. SIAM.

Du, I., and M. Marrone. 1992. "A proposal for user level sparse BLAS". Technical report, Rutherford Appleton Laboratory, Oxfordshire and CERFACTS, Toulouse and IBM Semea, Cagliari.

Duff, I. S. 1981a. "Algorithm 575: Permutations for a zero-free diagonal [F1]". *ACM Trans. Math. Software* vol. 7 (3), pp. 387–390.

Duff, I. S. 1981b. "On algorithms for obtaining a maximum transversal". *ACM Trans. Math. Software* vol. 7 (3), pp. 315–330.

Duff, I. S., and J. Koster. 1999. "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices". *SIAM J. on Matrix Analysis and Applications* vol. 20 (4), pp. 889–901.

Duff, I. S., and J. K. Reid. 1996. "The design of MA48: a code for the direct solution of sparse unsymmetric linear systems of equations". *ACM Trans. Math. Software* vol. 22 (2), pp. 187–226.

F., J. 2009. "Intel Math Kernel Library. Reference Manual". Technical report, Intel Corporation, Santa Clara, USA, 630813-054US.

Geiger, A., P. Lenz, C. Stiller, and R. Urtasun. 2013. "Vision meets Robotics: The KITTI Dataset". *Intl. J. of Robotics Research*.

George, A., and J. Liu. 1989. "The evolution of the minimum degree ordering algorithm". *SIAM Rev.* vol. 31 (1), pp. 1–19.

Gaël Guennebaud and Benoît Jacob and others 2010. "Eigen v3". http://eigen.tuxfamily.org.

Gustavson, F. G. 1978. "Two fast algorithms for sparse matrices: Multiplication and permuted transposition". *ACM Trans. Math. Software* vol. 4 (3), pp. 250–269.

Heggernes, P., S. Eisestat, G. Kumfert, and A. Pothen. 2001. "The computational complexity of the minimum degree algorithm". Technical report, Technical Report No. ICASE-2001-42, Institute for Computer Applications in Science and Engineering.

Kaess, M., A. Ranganathan, and F. Dellaert. 2007, April. "iSAM: Fast Incremental Smoothing and Mapping with Efficient Data Association". In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1670–1677. Rome, Italy.

Kümmerle, R., G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. 2011, May. "g2o: A General Framework for Graph Optimization". In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*. Shanghai, China.

Li, X., J. Demmel, J. Gilbert, iL. Grigori, M. Shao, and I. Yamazaki. 1999. "SuperLU User's Guide". Technical report, Technical Report No. LBNL-44289, Lawrence Berkeley National Laboratory.

Li, X. S., and J. W. Demmel. 2003. "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems". *ACM Trans. Math. Software* vol. 29 (2), pp. 110–140.

Liu, J. W. 1985. "Modification of the minimum-degree algorithm by multiple elimination". *ACM Trans. Math. Software* vol. 11 (2), pp. 141–153.

Lourakis, M., and A. Argyros. 2004. "The design and implementation of a generic sparse bundle adjustment software package based on the Levenberg-Marquardt algorithm". Technical report, Technical Report 340, Institute of Computer Science-FORTH, Heraklion, Crete, Greece.

Polok, L., V. Ila, and P. Smrž. 2013. "Cache Efficient Implementation for Block Matrix Operations". In *Proc. of the High Performance Computing Symp.*, pp. 698–706, ACM.

Polok, L., V. Ila, M. Šolony, P. Smrž, and P. Zemčík. 2013. "Incremental Block Cholesky Factorization for Nonlinear Least Squares in Robotics". In *Robotics: Science and Systems (RSS)*.

Polok, L., M. Šolony, V. Ila, P. Zemčík, and P. Smrž. 2013. "Efficient Implementation for Block Matrix Operations for Nonlinear Least Squares Problems in Robotic Applications". In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. IEEE.

Saad, Y. 1994. "SPARSKIT: a basic tool kit for sparse matrix computations–Version 2". Technical report, Computer Science Department, Univ. of Minnesota, Minneapolis, MN.

Schenk, O., and K. Gärtner. 2006. "On fast factorization pivoting methods for sparse symmetric indefinite systems". *etna* vol. 23 (1), pp. 158–179.

Strecha, C., W. von Hansen, L. Van Gool, P. Fua, and U. Thoennessen. 2008. "On benchmarking camera calibration and multi-view stereo for high resolution imagery". In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–8. IEEE.

AUTHOR BIOGRAPHIES

LUKAS POLOK was born in Brno (Czech Republic). Lukas received a MSc in Computer Science with a specialization in computer graphics and multimedia at Brno University of Technology, where he is presently employed as a researcher, working on efficient linear algebra algorithms using GPUs for general purpose calculations. His email address is ipolok 'at' fit.vutbr.cz.

PAVEL SMRZ is an associate professor in the Department of Computer Graphics and Multimedia, Faculty of Information Technology, Brno University of Technology, Czech Republic, where he leads the Knowledge Technology Research Group. His research interests include big data processing, hardware-accelerated machine learning, large-scale distributed and parallel processing, human-computer interaction, and information extraction. His email address is smrz 'at' fit.vutbr.cz.