Cache Efficient Implementation for Block Matrix Operations

Lukas Polok, Viorela Ila, Pavel Smrz
{ipolok,ila,smrz}@fit.vutbr.cz
Brno University of Technology, Faculty of Information Technology.
Bozetechova 2, 612 66 Brno, Czech Republic

Keywords: Block matrix, high performance, sparse BLAS, nonlinear least squares

Abstract

Efficiently manipulating and operating on block matrices can be beneficial in many applications, among others those involving iteratively solving nonlinear systems. These types of problems consist of repeatedly assembling and solving sparse linear systems. In the case of very large systems, without a careful manipulation of the corresponding matrices, solving can become very time consuming.

This paper proposes a memory storage scheme convenient for both, numeric and structural matrix modification and, at the same time, allowing efficient arithmetic operation. This scheme was used in the implementation of a simple BLAS-like library. The advantage of the new scheme is demonstrated through exhaustive tests on the popular University of Florida Sparse Matrix Collection. Furthermore, this library was used in solving several nonlinear graph optimization problems.

1. INTRODUCTION

Many applications ranging from physics, computer graphics, computer vision to robotics rely on efficiently solving large nonlinear systems of equations. In this case the solution can be approximated by incrementally solving a series of linearized problems. In some applications, the size of the system considerably affects performance. The most computationally demanding part is to assemble and solve the linearized system at each iteration. This paper exploits both, the block structure and the sparsity of the corresponding system matrices and offers very efficient solutions to manipulate, store and perform arithmetic operations.

A *block matrix* is a matrix which is interpreted as partitioned into sections called blocks that can be manipulated at once. A matrix is called *sparse* if many of its entries are zero. Considering both, the block structure and the sparsity of the matrices can bring important advantages in terms of storage and operations.

Using *dense* blocks is a natural way to minimize cache misses, since the CPU automatically prefetches the data as they are accessed. Nevertheless, taking care about the layout of the individual blocks in memory is also very important in order to avoid cache misses at block boundaries, especially if the blocks are small. Finally, the compressed format the blocks are to be stored in, needs to be chosen carefully otherwise the handling of the blocks can easily outweigh the advantage of cache efficiency.

Most of the existing state of the art implementations of nonlinear solvers rely on sparse block structure schemes. In general, the block structure is maintained until the point of solving the linear system. Here is where CSparse [1] or CHOLMOD [2] libraries are used to perform the matrix factorization. The CSparse is the state of the art element-wise implementation of operations on sparse matrices.

The advantage of *element-wise* sparse matrix schemes is that the arithmetic operations can be performed efficiently. Compressed column storage (CCS) format [3] used in CSparse is an efficient way to store the sparse data in memory. The disadvantage of this format is its inability or impracticality to change a matrix structurally or numerically once it has been compressed. The *block-wise* schemes are complementary, their advantages include both easy numeric and structural matrix modification, at the cost of memory overhead and reduced arithmetic efficiency, speed-wise.

The well known element-wise CCS sparse matrix representation [3] is as efficient (sometimes even more efficient) as any block matrix structure, in case of operating on a set of structurally-different matrices. Iterative solvers, however, involve operating iteratively on matrices where large portions of the matrix structure do not change between the iterations. In such case, block matrix schemes can be very proficient, as they allow for modifying parts of the block structure as well as efficiently modifying the numeric content.

This paper proposes a fast and cache efficient data structure for sparse block matrix representation, which combines the advantages of element-wise and block-wise schemes. It enables simple matrix modification, be it structural or numerical, while also maintaining, and often even exceeding the speed of element-wise operations schemes. Another important advantage of the proposed scheme is the overall robustness of the structure, allowing for error-checking, resulting in easier usage and in more stable error-free code.

2. RELATED WORK

Several sparse linear algebra libraries are currently available. They range from implementation of basic routines to complete linear algebra solutions [1, 4–7]. Recently, more sophisticated libraries implementing routines for iterative non-

linear solvers were developed [8]. This work is concerned with the implementation and evaluation of kernel operations and storage, and it is particularly focused on matrices having a block structure. The operations we tackle are the building blocks for any nonlinear solver, and the performance of their execution is crucial.

Standard interfaces for various linear algebra packages proved to be very useful in the past. Perhaps the most used include the three levels of BLAS [4–6], basic linear algebra subprograms, containing simple operations on vectors and matrices, and LAPACK [7], containing additional factorization functions and other more advanced functionality. These interfaces were originally proposed for dense matrices only. In time, other implementations emerged, including implementations for sparse matrices. Few of the available libraries support sparse block matrix operations, however.

CSparse [9], developed by Tim Davis [1] is one of the most used sparse linear algebra libraries. It is written in pure "C" and its functions are also available through MATLAB interface. It is highly optimized in terms of run time and memory storage and it is also very easy to use. It implements most of BLAS and some of LAPACK functionality, it was therefore used as a reference for comparison with the algorithms proposed in this paper. As mentioned above, CSparse stores its matrices in compressed column format which is suitable for operation on matrices, or coordinate format for simple matrix specification. Functions to convert between the formats are provided.

NIST Sparse BLAS [10] is also written in "C" and its source codes are generated from a set of kernel templates. Although very fast, it only implements a limited subset of BLAS. Operations, such as product of two sparse matrices, are not implemented. It introduces two block matrix storage formats, constant block size (CBS) and variable block size (VBR) compressed row. These are similar to CSparse's compressed column format. Unlike CSparse, it does not define any structure to store the matrices nor does it implement functions for conversion between different storage formats. As a result, it is rather hard to use as the proposed block storage scheme is quite complex. To our knowledge, it is the only library with BLAS interface to support block matrices.

Note that there are also other high performance implementations of BLAS, especially the parallel ones, which were omitted since the parallelism is not a focus of this paper. However, the proposed scheme can be parallelized.

Ceres-solver [8] is a portable C++ library for solving large nonlinear least squares problems. Ceres-solver is very popular since it is used at Google to estimate the pose of Street View cars, aircrafts, and satellites; to build 3D models for PhotoTours; to estimate satellite image sensor characteristics, and more. Ceres-solver uses CSparse for most of the linear algebra operations. It contains an internal implementation of block matrix storage and supports a limited set of operations on it, essentially the matrix-vector product. This block matrix functionality is not exported by the library, and as such is not available to the user out of the box. The block matrices in Ceres are stored in a way, similar to the scheme we propose in this paper, but their implementation does not allow for matrix modification and everytime the block matrix changes structurally, it needs to be rebuilt. This is a major drawback for the iterative nonlinear solvers as a significant amount of time will be lost in rebuilding the system matrix at every iteration.

The remainder of the paper is structured as follows. The next section introduces nonlinear least squares problem as the motivation of this work. Section 4. details the proposed implementation of the kernel operations. Section 5. shows the performance of the proposed solutions through benchmarks and time comparisons with the exiting implementations. Conclusions and future work are given in Section 6.

3. MOTIVATION

Nonlinear least squares (NLS) estimation is used in a broad range of applications across science and engineering. The basis of solving NLS problems is to approximate the nonlinear system by a linear one and to refine the parameters by successive iterations. Given the NLS problem:

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbf{R^n}}{\operatorname{argmin}} S(\mathbf{x}) \tag{1}$$

$$S(\mathbf{x}) = \sum_{i=1}^{m} r_i^2(\mathbf{x}) = \| \mathbf{r}(\mathbf{x}) \|^2 , \; m \geq n , \tag{2}$$

minimizes the sum of m squared nonlinear residuals $\mathbf{r}(\mathbf{x}) = [r_1(\mathbf{x}), ..., r_m(\mathbf{x})]^\top$, n being the number of variables.

Iterative methods such as Gauss-Newton or Levenberg-Marquardt are often used to solve the nonlinear problem. They are based on a series of linear approximations of $\mathbf{r}(x)$ around the current linearization point $\mathbf{x}^i$:

$$\tilde{\mathbf{r}}(\mathbf{x}^i) = \| \mathbf{r}(\mathbf{x}^i) + J(\mathbf{x}^i)(\mathbf{x} - \mathbf{x}^i) \|^2 , \tag{3}$$

where J is the Jacobian matrix which gathers the derivative of the components of $\mathbf{r}(\mathbf{x}_i)$. Gauss-Newton methods compute the correction $\delta = \mathbf{x} - \mathbf{x}^i$ by solving the linear least squares problem:

$$\delta^* = \underset{\delta}{\operatorname{argmin}} \| \mathbf{r}(\mathbf{x}^i) + J(\mathbf{x}^i)\, \delta \|^2 . \tag{4}$$

The solution δ^* can be obtained by solving the linear system. At each iteration, the Jacobian matrices $J(\mathbf{x}^i)$ calculated using the current linearization point $\mathbf{x}^i$, can be collected into the matrix A and the current residuals, into the right-hand side vector $\mathbf{b} = -\mathbf{r}(\mathbf{x}^i)$, to obtain the linear system in δ:

$$A\,\delta = \mathbf{b} . \tag{5}$$

The system matrix A can become very large in case of $m \gg n$. The normalized system has the advantage of remaining of the size of number of variables, n, even if the number of constraints, m, increases:

$$\Lambda \delta = \eta \,, \tag{6}$$

Here $\Lambda = A^T A$ is the normal system matrix and $\eta = A^T \mathbf{b}$ is the corresponding right hand side.

This paper is concerned with system matrices, A or Λ, which are sparse and have an underlying block structure, where the size of the blocks corresponds to the number of degrees of freedom of the variables. The solution of the linear system can be obtained either using sparse matrix decompositions, QR for the system in (5) or Cholesky for the system in (6), or gradients methods.

The process of assembling and solving very large, sparse linear systems can become very expensive when the size of the problem grows. The used data structure has to allow both, efficiently re-building the system every time a new linearization point is available and high speed arithmetic operations. The reminder of the paper introduces a solution to this problem, which highly exploits the matrix sparse block structure.

4. PROPOSED IMPLEMENTATION

When dealing with matrices with a block structure, operating on dense blocks is a natural way to support vectorization and improve cache efficiency without any additional effort. Note that this only holds for SIMD type processors, and likely would not be practical for true vector processors, such as Cray machines, where interleaved block storage would be more beneficial. On the other hand, the use of dense blocks allows efficient data representation at their natural granularity, making it simple to reference the data inside the matrix and change their value when it is needed.

In the existing block matrix implementations, the blocks are usually allocated on the heap, and it can not be guaranteed that the blocks are allocated in close memory locations. If the blocks are allocated in distant memory locations, cache misses still occur. To improve that, our implementation uses segregated storage, which guarantees that the blocks are stored tightly next to each other.

The arithmetic efficiency of block matrices is mostly reduced, compared to element-wise sparse matrices. That is because two extra loop counters for block rows and block columns are needed. Our implementation elegantly solves this issue using metaprogramming. Since for the least square problems the size of the blocks corresponds to the number of degrees of freedom of the variables, the possible block sizes of a given problem are known in advance at compile time. It is therefore possible to hint the individual operations on matrices with lists of possible block sizes occuring in the operands.

4.1. The data structure

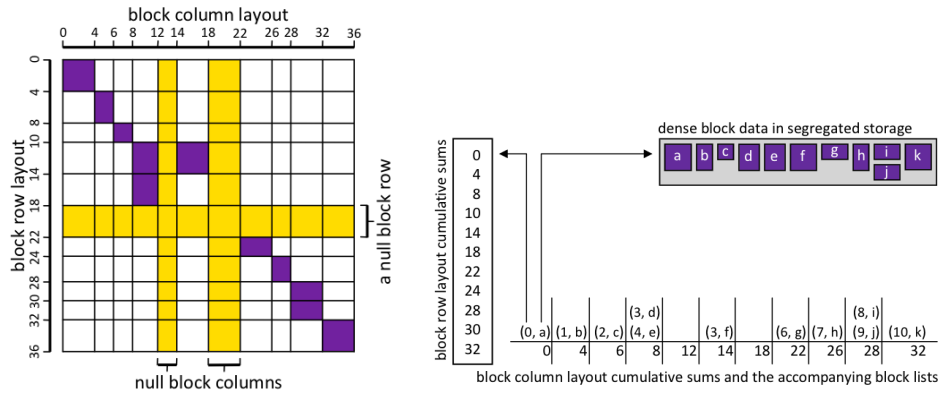
In general, all existing block matrix schemes, including ours, involve the same data layout as CCS representation (or equivalent), but use more complicated data structures to make the matrix structure editable. For example, in some existing implementations [8], trees or other higher abstract data structures are used.

In the proposed block matrix implementation, block row and block column layouts are described using the same structure, except that the columns also contain the non-zero matrix blocks. The structure is implemented as a sorted list of cumulative sums of block sizes (see Fig. 1 a)). The matrix blocks are also stored in a sorted list. Each matrix block contains row index and a pointer to matrix data. The data itself is allocated in forward-allocated segregated storage (see Fig. 1 b)), a storage model similar to a pool but only permitting allocation and de-allocation of elements from the end of the storage, in the same manner stacks do. This yields fast allocation and improves cache coherency.

The choice of a sorted list over e.g. a tree structure is given by the nature of matrix usage. When iteratively solving an NLS problem, the block columns or block rows are created once and used (referenced) many times. This reflects the nature of a sorted list where insertion is costly (except for the insertion at the end) but lookup is fast. At the same time the flat structure is cache friendly. Tree structures have more balanced insertion and lookup costs, but since the nodes of a tree are typically allocated on the heap, cache misses are incurred at every lookup.

In order to enable the unusually fast $O(1)$ block lookup in arithmetic operations and also to facilitate error checking for incorrectly placed blocks, one important restriction on block and column layouts must be applied. The whole area of the matrix needs to be represented, which means that the layout of null rows or columns needs to be represented as well. Those are marked in yellow in Fig. 1 a) and their representation is shown in Fig. 1 b) where the fifth and sixth fields from the block column layout are empty.

This contrasts with the usual sparse block matrix representations, which only describe the layout of nonzero blocks without caring about the null elements in between. It comes at the cost of small increase in memory requirements, but only for the layout itself, not for the data. If n_b and m_b are the number of block rows and columns, respectively, up to $floor(m_b/2) + floor(n_b/2)$ additional cumulative sums are stored in the worst case. These describe the layout of null block rows and columns. This assumes no null space fragmentation which indeed does not occur in our implementation. The exact amount of required extra memory depends on the positions of the nonzero blocks in the matrix. Please note that for NLS problems there are no such null columns or rows, therefore, no extra space requirements apply.



(a) (b)

Figure 1. Block row/column layout of a block matrix. a) An example of a sparse block matrix and the actual values of the cumulative block sum (on top and left side). Non-zero dense blocks are shown in violet. Yellow shows null rows/columns. b) Dense block data in segregate storage. On the bottom, we show the block column layout and the corresponding sorted list of pairs of type (iRL, pDB), where iRL is the index of the row layout, and pDB is the pointer to the block data in the memory.

4.2. Block matrix insertion

In order to write (scatter) a block in the matrix, the block column and block row need to be resolved first. Adding a new block row or column inside the matrix area, or reusing or subdividing an existing one is a *logarithmic time* operation. However, incrementally appending the matrix with blocks to or after the last block row or column is a *constant time* operation, as it only needs to determine whether to create a new block row or column at the end, or to use an existing one. This is a basic operation but frequently used in the context of incremental solvers where the system matrix grows every step.

In order to lookup a block by its position given by the row and column numbers in elements, the block column and block row are resolved first in $O(\log n_b + \log m_b)$ time. Then the block needs to be found in the sorted list, taking additional $O(\log f_b)$ time (f_b being the number of nonzero blocks, the fill-in, of a given column; for most sparse matrices $f_b \ll m_b$). This operation can mostly be avoided by storing a reference to the block after inserting it in the matrix. This is very useful for updating the system matrices in (5) or (6) every time a new linearization point is available. In this case, the new values of the blocks can be calculated directly inside the matrix, avoiding data copying or block lookup. In addition, our implementation allows insertion of block using logical indexing, where the block position is given by indices of block row and block column. That avoids the block column and row resolution and only requires to find the block in a sorted list, taking $O(\log f_b)$ time. This feature is useful for applications that insert many blocks in the same column, and for arithmetic operations which operate with logical indexing.

$C \leftarrow \text{MULT}(A, B)$

```
 1: C ← blank-matrix(rows(A), cols(B))
 2: for each columnB_block in B do
 3:     colB ← column-of(columnB_block)
 4:     for each blockB in columnB_block do
 5:         rowB ← row-of(blockB)
 6:         columnA_block ← find-column(rowB, A)
 7:         for each blockA in columnA_block do
 8:             rowA ← row-of(blockA)
 9:             block_dest ← alloc-block(rowA, colB, C)
10:             block_dest ← blockA * blockB
11:         end for
12:     end for
13: end for
```

Algorithm 1: Block Matrix Multiplication.

4.3. Arithmetic operations

The arithmetic operations on block matrices are carried out in the same manner as on element-wise sparse matrices, with the exception of handling matrix blocks instead of scalar values. Most of the arithmetic operations require block lookup at some point. In other existing block matrix implementations, the $O(\log n_b)$ lookup is used, and an example of the matrix multiplication is given in Algorithm 1. On line 6, an $O(\log n_b)$ lookup is required to find block column. Then on line 9, another $O(\log n_b + \log m_b + \log f_b)$ lookup is performed in order to place a new block in the destination matrix. To improve performance, a mapping function can be used. Consider Algorithm 2. First, note the use of logical indexing of block rows and block columns by their id, instead of by their physical position in elements, used in Algorithm 1. The mapping is calculated as a projection from block rows of the B ma-

```
C ← FastMult(A, B)
 1: C ← blank-matrix(rows(A), cols(B))
 2: f_map ← mapping-function(block-cols(A)), block-rows(B))
 3: colB_id ← 0
 4: for each columnB_block in B do
 5:     for each blockB in columnB_block do
 6:         rowB_id ← row-id-of(blockB)
 7:         columnA_id ← f_map(rowB_id)
 8:         if columnA_id == null then
 9:             exit {block layout mismatch, product not defined}
10:         end if
11:         columnA_block ← block-cols(A)[columnA_id]
12:         for each blockA in columnA_block do
13:             rowA_id ← row-id-of(blockA)
14:             block_dest ← alloc-block-log(rowA_id, colB_id, C)
15:             block_dest ← blockA * blockB
16:         end for
17:     end for
18:     colB_id ++
19: end for
```

Algorithm 2: Fast Block Matrix Multiplication.

trix to block columns of the A matrix. The cost of calculating the mapping function is $O(n_b)$ in the number of block rows or block columns. Note that the mapping function needs to be only calculated once, before the arithmetic operation takes place. Also, the complexity involved is negligible, compared to the complexity of the arithmetic operation itself. This later allows to replace the logarithmic time $columnA_{block}$ lookup by an $O(1)$ lookup. It also enables checking whether the matrix product is defined on the given block matrices. Further, the insertion of the block only requires insertion into the sorted list which is $O(\log f_b)$ but avoids lookup of block row and block column. Note that for some types of operands (such as diagonal matrices or symmetric matrices), the order of inserted blocks can be anticipated and the $O(\log f_b)$ time lookup can be avoided. In our implementation, this is used to optimize matrix products in the A^TA form.

As mentioned above, the block sizes correspond to the DOF of the variables and, in general, are known in advance. Using typelists and templates, decision trees are built at compile time that later at runtime enable the use of code generated for a given block size. This allows for optimization using loop unrolling and vectorization at the block level, e.g. in Algorithm 2 at line 15. It can be easily shown that if log_2 of the number of possible block sizes is smaller than the average block size, the resulting code will contain less branching and thus will run faster. Note that in C++, this functionality is accessible using simple and easy to read syntax where the list of block sizes is passed to each individual matrix operation call in angled brackets. It would also be possible to restrict whole matrices to only contain blocks of specified sizes, but this solution was seen as less versatile, and was not implemented.

5. PERFORMANCE ANALYSIS

In this section we compare the timing results for several matrix operations performed using the proposed implementation with similar state of the art implementations such as CSparse, Ceres and NIST Sparse BLAS. Note that aparts from the latter two, at the time of writing the paper, we were not aware of any other practical block matrix implementations. NIST implementation can store matrices in several formats. CSR is a compressed sparse row element-wise format, similar to the one used in CSparse. BSR denotes constant block size compressed block row format, and is a simple block matrix format where all the blocks have the same size. Finally, VBR denotes variable block size compressed block row format, which is an extension of BSR where the individual blocks can have arbitrary size. This format is the most general, and is equivalent to the one used in Ceres and by the proposed solution. The proposed implementation is denoted as UBlock, and the version with metaprogramming optimization is denoted UBlock FBS (fixed block size).

In the second section of this chapter, we also briefly discuss speed improvements on an incremental NLS implementation. For more comperhensible comparison, see [11].

All the tests were performed on a computer with Intel Core *i*5 CPU 661 running at 3.33 GHz and 4 GB of RAM. This is a quad-core CPU without hyperthreading and with full SSE instruction set support. During the tests, the computer was not running any time-consuming processes in the background. Each test was run ten times and the average time was calculated in order to avoid measurement errors, especially on smaller matrices. The computer was running Ubuntu 11.10 (64 bit) and all the tested libraries were compiled using g++ version 4.6.1.

5.1. Matrix Operations Performance

The evaluation was performed on a subset of the The University of Florida Sparse Matrix Collection [12]. This collection was chosen because it contains sparse matrices corresponding to a diverse set of problems, and as such it is suitable for testing of general purpose linear algebra implementations. However, this collection does not contain any block matrices, and for that reason only the structure of the matrices was used, and each nonzero element was assumed to be a block of size given by each particular test configuration. As the speed of blockwise operations depends on block size, the block size was varied from 1×1 to 30×30 elements. Note that these benchmarks are synthetic, but still highly relevant in the context of problems with natural block structure, such as (but not limited to) NLS.

Several matrices were selected for comparison. In particular, the MCCA matrix, a relatively small matrix of 180×180 elements containing 2659 nonzero entries, was used for the comparison with the NIST implementation. This matrix was

selected because the authors already performed experimental evaluation [10] on it. Their block matrix representation is rather complex and it would present a significant effort to compare also on other matrices. Since the NIST implementation is not widely used, this limited comparison should be sufficient. For the rest of the comparisons, 200 matrices from The University of Florida Sparse Matrix Collection were chosen randomly.

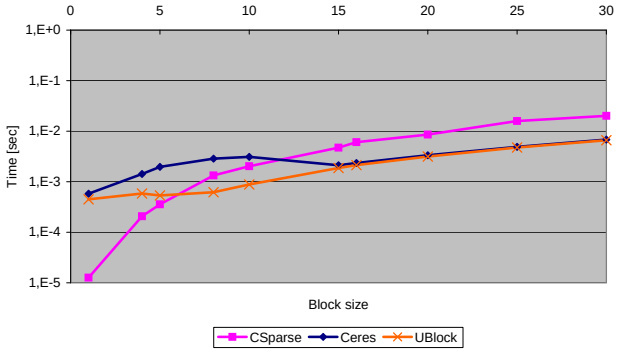


Figure 2. Time for compression of the MCCA matrix (smaller is better).

A comparison of the time required to compress a sparse matrix using CSparse, Ceres and our implementation is shown in Fig. 2. The NIST implementation is missing from the plot because their library does not provide compression routines. Note that CSparse time is directly dependent on the number of matrix nonzero elements. The block schemes become more efficient as the block size grows; our implementation becomes the fastest for 6×6 blocks (or larger). The test was run on the MCCA matrix [12].

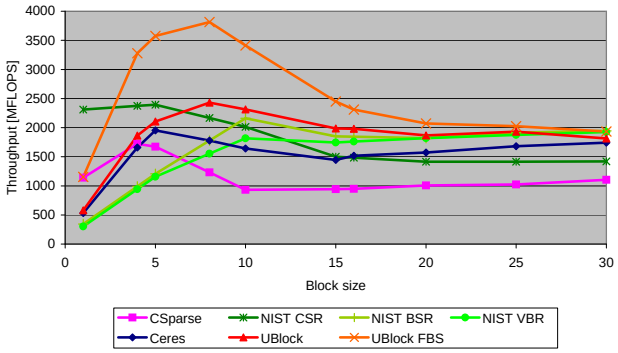


Figure 3. General matrix vector product on the MCCA matrix.

Similarly, Fig. 3 shows the time comparison for the general matrix vector product operation. For 1×1 blocks, CSparse is faster than every other implementation, except for the NIST element-wise implementation and the proposed fixed block size implementation. Although the NIST element-wise implementation is very fast and significantly outperforms CSparse, there is only small speedup with their block matrix formats. For block size 1×1, the NIST element-wise sparse implementation is the fastest. Interestingly enough, the Ceres implementation is slower than the NIST implementation, approaching NIST performance as the block size grows. It becomes faster than CSparse for block size 5×5. Our general implementation becomes faster than CSparse for 4×4 block and is the fastest for 8×8 blocks or larger. However, the proposed fixed block size implementation is always the fastest. Note that for 1×1 blocks, it is even slightly faster than the CSparse library.

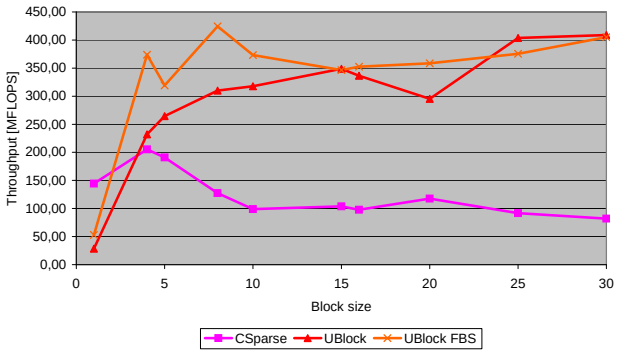


Figure 4. Linear combination of the MCCA matrix and its transpose.

An additional benchmark is performed for the operation of addition of the matrix and its transpose. This operation is not particularly important in the context of nonlinear solvers, but due to its arithmetic simplicity it is sensitive to efficient data manipulation. Since the MCCA matrix is not structurally symmetric, the result of this operation has a different nonzero pattern than the operands. That can be expected in most matrix addition situations, therefore it serves as a valid benchmark. The results can be seen in figure 4. Note that the time spikes of the proposed implementation, especially on the fixed-block-size version, are caused by the compiler being able to generate SSE optimized code for blocks of sizes that are multiples of four.

Multiplication benchmark in Fig. 5 displays similar behaviour. Note that the gap between element-wise sparse and blockwise sparse implementation gets very wide as the block size increases. On the other hand, most of the popular nonlinear least squares problems will likely only use blocks up to no more than 10×10.

We also performed cache profiling using the Cachegrind tool, with the default settings (64 kB of L1 cache and 6 MB of L2 cache). The benchmark with the MCCA matrix was run several times in order to identify outliers in Cachegrind results. The test was run with block size 4×4, and confirmed that the proposed storage is indeed cache efficient.

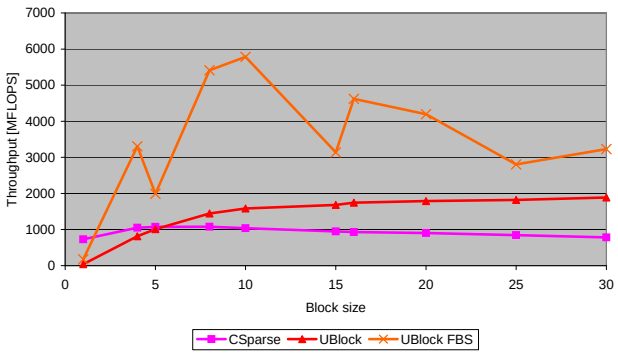


Figure 5. Product of the MCCA matrix and its transpose.

Matrix multiplication had 8.3 % L1 cache misses and 16.3 % last level cache misses, compared to CSparse. Similarly, matrix vector multiplication reduced L1 cache misses down to 14.2 % and last level cache misses to 9.45 %.

Additional benchmarks are shown in Table 1, which contains the average run times on 200 randomly chosen matrices from The University of Florida Sparse Matrix Collection [12]. The benchmarks involved matrix addition, matrix product, optimized matrix product for symmetric matrices, matrix - vector product, matrix compression from sparse values in triplet form, matrix transpose and the triangular solve operation. Note that some of the above operations could only be executed on a subset of chosen matrices. It can be seen that for 1×1 blocks, CSparse is the fastest, except for the triangular solve operation. Otherwise the proposed implementation consistently yields better times, with the fixed block size optimization being faster than the general optimization. The only exception is the compression benchmark, where Ceres also gets good results. This is understandable as Ceres does not provide any functionality to change the matrix once it has been compressed, which makes the storage simpler. This is a disadvantage in the context of incremental iterative solvers, since the system matrix adds a few new blocks at every step and it is considerably more efficient to have an option to alter compressed matrix than to recompress at every step. For performance evaluation in a real NLS application, please refer to [11]. Also note that the proposed scheme only accelerates problems with inherent block structure, and is not suitable for general sparse matrix operations where CSparse is faster.

5.2. Solving NLS Performance

In order to evaluate our new efficient block matrix scheme, we implemented a standard nonlinear graph optimization algorithms based on Gauss-Newton iterative solver. The tests were performed on graphs of different sizes (from $3.5 \cdot 10^3$ to $100 \cdot 10^3$ variables), different sparsity patterns and different disparity between number of variables and constraints (n and m). A detailed performance analysis and comparison with other existing NLS solver implementations was presented in [11]. Here we highlight some of the achievements. Our implementation outperforms all the tested implementations in both batch and incremental mode. The comparison in batch mode shows speed up of 10% when compared to the fastest implementation. This is mainly due to the proposed block matrix scheme, the algorithm being very similar and the differences in the implementation style cannot cause large speedups. Furthermore, in incremental mode, where the system is solved every time a new constraint is integrated, the efficient matrix operations start paying off, as there is a larger portion of time spent in updating the system. The proposed implementation shows speed up of more than 50%. The implementation is available as open source at `http://sourceforge.net/p/slam-plus-plus/`.

6. CONCLUSIONS AND FUTURE WORK

A new implementation for block matrix operations was proposed in this paper. It implements highly efficient kernels that are core for nonlinear least squares solvers. We targeted problems that have a particular block structure, where the size of the blocks corresponds to the number of degrees of freedom of the variables.

The proposed scheme combines the advantages of *block-wise* schemes convenient in both, numeric and structural matrix modification and *element-wise*, which are efficient in arithmetic operations. It also allows to conveniently restrict possible block sizes to a defined set (per every instance of matrix operation), at compile time. This leads to further substantial speedup. The advantage of the new scheme was demonstrated through comparisons with the existing implementations on a subset of matrices from University of Florida Sparse Matrix Collection dataset.

Even though the proposed scheme proved to outperform the state-of-the-art implementations, several improvements from algorithmic point of view can be applied. Support for special matrix types, such as diagonal or band-diagonal and symmetric matrices can be provided. Furthermore, some of the block matrix operations can be efficiently parallelized. The block layout was designed with hardware acceleration in mind. This is important for large problems, which can efficiently run on wide scale of accelerators, from multicore CPUs to clusters of GPUs.

7. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union, 7^{th} Framework Programme grants 316564-IMPART and 247772-SRS, Artemis JU grant 100233-R3-COP, and the IT4Innovations Centre of Excellence, grant n. CZ.1.05/1.1.00/02.0070, supported by Operational Programme Research and Development for Innovations funded by Structural Funds of the European Union and

			Block Size						
			1×1	4×4	5×5	8×8	10×10	15×15	16×16
Benchmark	**Library**	**Mode**	**Time** [ms]						
Matrix Add	CSparse		**0.101**	1.497	2.574	7.232	12.081	26.877	31.890
	UBlock		0.389	0.896	1.261	2.747	4.048	7.884	8.934
		FBS	0.198	**0.586**	**0.939**	**2.500**	**3.785**	**7.438**	**8.546**
Matrix Product	CSparse		**0.672**	23.079	42.608	144.294	271.700	908.861	1096.108
	UBlock		11.601	24.555	37.316	86.752	148.873	421.273	495.385
		FBS	3.330	**8.440**	**20.506**	**31.895**	**55.363**	**261.459**	**242.498**
Symmetric Matrix Product	UBlock		4.821	12.256	18.360	49.159	85.476	257.207	310.401
		FBS	4.966	9.014	15.212	24.284	65.773	146.110	239.969
Matrix Vector Product	CSparse		**0.012**	0.204	0.357	1.018	1.550	3.237	3.643
	Ceres		0.031	0.165	0.247	0.646	0.992	2.083	2.280
	UBlock		0.028	0.148	0.238	0.625	0.962	1.890	2.153
		FBS	0.016	**0.107**	**0.185**	**0.556**	**0.931**	**1.706**	**1.999**
Compress	CSparse		**0.037**	0.851	1.490	4.266	6.916	15.001	18.480
	Ceres		0.530	**0.815**	**1.049**	**2.062**	2.906	5.494	6.378
	UBlock		1.167	1.380	1.487	2.211	**2.844**	**5.152**	**5.767**
Transpose	CSparse		**0.040**	0.787	1.348	4.223	7.080	18.625	24.474
	UBlock		0.337	**0.639**	**0.854**	**1.629**	**2.497**	**5.054**	**5.817**
Triangular Solve	CSparse		0.015	0.168	0.279	0.823	1.305	2.976	3.463
	UBlock		0.024	0.126	0.190	0.500	0.752	1.661	1.877
		FBS	**0.014**	**0.089**	**0.155**	**0.455**	**0.661**	**1.472**	**1.763**

Table 1. Timing results on a subset of University of Florida Sparse Matrix Collection [12]; the best times are in bold.

the state budget of the Czech Republic.

REFERENCES

[1] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, 2006.

[2] T. A. Davis and W. W. Hager, "Modifying a sparse cholesky factorization," 1997.

[3] Y. Saad, *SPARSKIT: A basic tool kit for sparse matrix computation*. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990.

[4] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[5] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, "An extended set of basic linear algebra subprograms: model implementation and test programs," *ACM Transactions on Mathematical Software (TOMS)*, vol. 14, no. 1, pp. 18–32, 1988.

[6] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, *et al.*, *LAPACK Users' guide*, vol. 9. Society for Industrial Mathematics, 1987.

[8] S. Agarwal and K. Mierle, "Ceres solver." `http://code.google.com/p/ceres-solver/`, 2012.

[9] T. Davis, "Csparse." `http://www.cise.ufl.edu/research/sparse/CSparse/`, 2006.

[10] S. Carney, M. Heroux, G. Li, and K. Wu, "A revised proposal for a sparse blas toolkit," 1994.

[11] L. Polok, S. V. Ila, M. Solony, P. Zemcik, and P. Smrz, "Efficient implementation for block matrix operations for nonlinear least squares problems in robotic applications," in *Proceedings of the IEEE International Conference on Robotics and Automation*, IEEE, 2013.

[12] T. Davis, "The university of florida sparse matrix collection," in *NA digest*, Citeseer, 1994.