High-Performance Graphics 2019
T. Foley and M. Steinberger
(Guest Editors)

Volume 38 (2019), Number 8

An Efficient Solution to Structured Optimization Problems using Recursive Matrices

D. Rückert[1] and M. Stamminger[1]

[1]University of Erlangen-Nuremberg, Germany

Abstract

We present a linear algebra framework for structured matrices and general optimization problems. The matrices and matrix operations are defined recursively to efficiently capture complex structures and enable advanced compiler optimization. In addition to common dense and sparse matrix types, we define mixed matrices, which allow every element to be of a different type. Using mixed matrices, the low- and high-level structure of complex optimization problems can be encoded in a single type. This type is then analyzed at compile time by a recursive linear solver that picks the optimal algorithm for the given problem. For common computer vision problems, our system yields a speedup of 3-5 compared to other optimization frameworks. The BLAS performance is benchmarked against the MKL library. We achieve a significant speedup in block-SPMV and block-SPMM. This work is implemented and released open-source as a header-only extension to the C++ math library Eigen.

CCS Concepts

• ***Computing methodologies*** → *Symbolic and algebraic algorithms; Linear algebra algorithms; Optimization algorithms;*

1. Introduction

Many algorithms in visual computing apply numerical methods based on simple linear algebra procedures. A large number of libraries are available to solve such procedures efficiently. In many cases, the underlying matrices have a certain structure, e.g. they can be (tri-)diagonal, sparse, or block-sparse. Their usage is only practical (both in terms of performance and memory consumption), if this structure is exploited. To this end, typical BLAS implementations provide specialized kernels which are optimized for a specific matrix structure.

In this paper, we present a simple, but highly efficient method to better describe and exploit matrix structure using C++-templates. Our approach is centered around recursive matrix types and recursive basic linear algebra operations. A simple example is a block-sparse matrix, which can be described by a sparse matrix of dense matrices. More complicated recursive structures are also possible and will be used in this paper. We present an implementation of recursive matrices based on the *Eigen* library [GJ*10], but the approach can readily be generalized to other linear algebra frameworks. We show that recursive matrices are an easy-to-use and elegant way to define non-linear least squares problems, which also assist compilers in the generation of highly efficient code.

In the following, we describe the concept of recursive matrices and the necessary steps to include recursive types into existing linear algebra packages, in particular into *Eigen*. We show how to apply this approach to three standard problems of visual computing, namely pose graph optimization, bundle adjustment, and ARAP. In all three problems, our approach outperforms other common optimization frameworks by a factor of 3-5.

In summary, the contributions of this work are:

- A set of rules to create recursive algorithms from basic linear algebra code.
- The definition of *mixed matrices* and their application on structured non-linear least squares problems.
- A specialized linear solver that analyzes the matrix structure at compile time to generate problem-specific optimal code.
- An open-source extension [Rü19] to the Eigen math library for recursive matrix operations, which outperforms state-of-the-art BLAS and optimization frameworks.

2. Related Work

Linear algebra operations on sparse block matrices is a widely researched topic in the high-performance community. It has been shown that exploiting the block structure can significantly speed up matrix-vector [VM05, WOV*07] and matrix-matrix multiplication [BG08]. This speed up also transfers to high level algorithms. Esmond et al. [NP93] presented a block-sparse Cholesky factorization algorithms that outperforms all non-blocked implementations due to reductions in memory traffic and indirect indexing.

Furthermore, recursive basic matrix operations have received attention over the last decades. A famous example is the Strassen algorithm for matrix multiplication [Str69], which is still used on modern systems [HSHvdG16]. More recent work uses recursive

formulations to generate highly optimized code for specific architectures. Recursion allows for efficient utilization of a memory hierarchy that has the potential of matching every level of a deep cache hierarchy [EGJK04]. Numerous implementations are available that adjust the blocking-size to the processor and outperform any non-recursive implementations [GVDG08, AGK*00]. Demmel et al. [DEF*13] show how to apply recursive matrix multiplication on highly parallel workstations and super computers. It has been shown that recursion can speed up approximate factorization methods for block matrices on parallel processors [AP86]. Andersen et al. [AWG01] presented a recursive Cholesky factorization algorithm that relies on a special blocked data format for recursive matrices [GHJ*98]. A direct use-case of recursive data formats are hierarchical $\mathcal{H}$-matrices [Beb08]. On each level of the $\mathcal{H}$-matrix, the current block is subdivided in a quadtree-like pattern until a specific termination criterion is met. Leaf nodes are stored either as dense matrices G or as low-rank approximations $G = AB^T$. Depending on the compression rate, the matrix-vector product of $\mathcal{H}$-matrices can be computed in the complexity $O(N \log^q N)$. The FLAME library [VZCvdG*09] provides a LAPACK implementation that makes use of recursive algorithms to speed up the computations. FLAME also supports user-defined $\mathcal{H}$-matrices, but only for dense operations, which makes it not feasible for very sparse optimization problems.

Non-linear least squares optimization problems can be solved in various ways. A common method, which uses only first order derivatives, is the iterative Levenberg-Marquardt algorithm [Mor78]. General C++ implementations are available that can solve a wide range of such problems. G2O [KGS*11] is an optimization framework for non-linear least squares problems, which represents the unknowns as vertices in a graph and the residuals as edges. The Ceres-Solver [AMO12] implements similar functionality, but also supports automatic differentiation and provides a very user-friendly interface. Due to their simple interface, these frameworks introduce additional run-time overhead. For example, the structure of the optimization problem is analyzed at run-time and appropriate solvers must be picked by the user.

Domain-specific languages (DSLs) have been introduced to generate highly optimized code for a given problem group. The DSLs EBB [BSL*16] and Simit [KKRK*16] allow the user to express abstract linear algebra operations over graphs. In the case of EBB, the graph is generalized to arbitrary relations such as regular grids and hypergraphs. Both, EBB and Simit were build and optimized for solving partial differential equations using the finite element method. DeVito et al. [DMZ*17] have presented a specialized DSL for non-linear least squares problems. They are able to generate optimized GPU code only from the energy function. Similar to EBB, multiple predefined structures such as graphs and grids are supported. In contrast to these DSLs, our recursive approach is able to provide similar functionality without the need of a new language. Graphs, hypergraphs, and grids are all implicitly given by sparse and dense matrices. Optimized code is automatically generated due to specialization and template meta programming. Additionally, a recursive formulation allows the user to define a multi-level structure, which further increases cache locality and compile time optimizations.

3. Recursive Matrix Types

Almost all linear algebra operations can be defined recursively by changing the storage and execution order. As an example, we consider the matrix-vector product $y = Ax$ with $A \in \mathbb{R}^{4\times4}$ and $x, y \in \mathbb{R}^{4\times1}$. The corresponding C++ code using Eigen matrix types is:

```
Matrix<double,4,4> A;
Matrix<double,4,1> x, y;
// ...
y = A * x;
```

$$\begin{bmatrix} 3 & 5 & 1 & -2 \\ 0 & -1 & 5 & 10 \\ 3 & 5 & 1 & 9 \\ -2 & 1 & 6 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 4 \\ 7 \\ -3 \end{bmatrix} = \begin{bmatrix} 36 \\ 1 \\ 3 \\ -33 \end{bmatrix} \tag{1}$$

In this case, the matrix A is stored either row-major or column-major in memory. By changing the storage order so that each 2×2 block is stored consecutively, we obtain the following block-representation:

$$\begin{bmatrix} \begin{pmatrix} 3 & 5 \\ 0 & -1 \end{pmatrix} & \begin{pmatrix} 1 & -2 \\ 5 & 10 \end{pmatrix} \\ \begin{pmatrix} 3 & 5 \\ -2 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 9 \\ 6 & 6 \end{pmatrix} \end{bmatrix} \cdot \begin{bmatrix} \begin{pmatrix} 1 \\ 4 \end{pmatrix} \\ \begin{pmatrix} 7 \\ -3 \end{pmatrix} \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} 36 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 3 \\ -33 \end{pmatrix} \end{bmatrix} \tag{2}$$

It can easily be shown that (1) and (2) are semantically equivalent but translate to a different order of execution. The first row of (1) is evaluated to

$$((3\cdot1+5\cdot4)+1\cdot7)-2\cdot(-3) = ((3+20)+7)+6 = 36, \tag{3}$$

whereas the first row of Equ. (2) translates to

$$(3\cdot1+5\cdot4)+(1\cdot7-2\cdot(-3)) = (3+20)+(7+6) = 36. \tag{4}$$

The latter matrix-vector product (2) can be elegantly defined in C++ using recursive matrix types. For this example, the outer type and the inner type are both 2×2 dense matrices:

```
Matrix<Matrix<double,2,2>,2,2> A;
Matrix<Matrix<double,2,1>,2,1> x, y;
// ...
y = A * x;
```

The multiplication of these types automatically applies recursion, because the *-operator is overloaded for matrices as well as scalars. Unfortunately, other linear algebra expressions do not work in the Eigen math library directly, because the implementation implicitly assumes that the elements of matrices are scalars. In some cases, this assumption can work due to operator-overloading, but, for example, Eigen's dot-product generates a compiler error:

```
Scalar dot(Vector a, Vector b){
  Scalar sum = 0;
  for(int i = 0; i < a.rows(); ++i)
    sum += a(i) * b(i);
  return sum;
}
```

If we run this function with a vector of vectors (as used in the previous example), the compiler throws an error in line 4. Each element

of a and b are vectors, therefore this line results in an illegal vector-vector product. We can fix this, by transposing `a(i)` in line 4 and adding a transpose-specialization for scalar arguments:

```
Scalar transpose(Scalar a) { return a; }
Scalar dot(Vector a, Vector b){
  Scalar sum = 0;
  for(int i = 0; i < a.rows(); ++i)
    sum += transpose(a(i)) * b(i);
  return sum;
}
```

The remainder of this section defines general rules for recursive matrix frameworks. These rules must be applied to all implemented algorithms, if we want to use recursive matrix types without limitations. For each rule, we include one example from the Eigen library that violates it and present an alternative working implementation.

1. **Consistent Multiplicative Ordering**
   Since the general matrix multiplication is not commutative, the order must propagate recursively into inner expressions. For example, the diagonal times dense matrix multiplication $R = DA$ is implemented in Eigen as

   ```
   // For every i,j...
   R(i,j) = A(i,j) * D(i,i); // < Incorrect
   ```

   which only works for commutative scalar elements of A and B. To make it usable for recursive types, we have to swap the multiplication order:

   ```
   // For every i,j...
   R(i,j) = D(i,i) * A(i,j);
   ```

2. **Augmented Scalar Assignments**
   Scalar assignments of the form $A = \alpha$, where A is an unknown recursive type and α is a scalar must be replaced by a more descriptive expression. If A is used in an additive context, each element of A is set to α. Therefore, the scalar assignment `sum=0` generates the neutral element of the additive group. If A is used in a multiplicative context, A is initialized with an identity matrix scaled by α. For the scalar assignment `factor=1`, this creates the neutral element of the multiplicative group. In the Eigen library, the matrix-vector product $y = Ax$ computes each element of y with

   ```
   y(i) = 0;  // < Compile Error
   // ...
   y(i) += A(i,j) * x(j);
   ```

   The assignment `y(i)=0` does not work, if `y` is a vector of vectors. Since `y(i)` is used in an additive expression later, we use the helper function `assign_plus` that initializes all elements of `y(i)` to 0:

   ```
   y(i) = assign_plus<T>(0);
   // ...
   y(i) += A(i,j) * x(j);
   ```

3. **Transpose Propagation**
   The matrix transpose operation must be propagated recursively to the inner types. This includes the explicit transpose computation $B = A^T$ and implicit transpositions $x = C^T y$. The latter expression is called implicit, because C^T is not constructed to evaluate the product $C^T y$. For example, in Eigen's transposed-matrix-vector product $y = A^T x$

   ```
   // ...
   y(j) += A(i,j) * x(i);
   //      ^^^^^^ Incorrect
   ```

   the transposition must be propagated to all elements `A(i,j)` before multiplying with `x(i)`:

   ```
   // ...
   y(j) += transposeRef(A(i,j)) * x(i);
   ```

4. **Extended Basic Operators**
   Some high-level algorithms use operators that are usually defined only for scalars. These operators must be extended to work on recursive types. This includes division $C = A/B$ and square root $\sqrt{A}$. The division of non-scalar types is given by $A/B = B^{-1}A$, where B^{-1} is the matrix inverse. The square root is defined as $\sqrt{A}\sqrt{A} = A$. For recursive matrix types, a positive square root exists, if the matrix is positive semi-definite. For example, in the LDLT factorization, the elements of L are divided by the diagonal element:

   ```
   // ...
   tmp = L(k,i) / D(i);  // < Compile Error
   ```

   This results in a compile error, because the division operator is not defined for arbitrary types. Instead, we compute the inverse of `D(i)` and multiply it left-handed to `L(k,i)`:

   ```
   // ...
   tmp = inverse(D(i)) * L(k, i)
   // ...
   // + Additional specialization for scalars
   Scalar inverse(Scalar s) { return 1 / s; }
   ```

4. Structured Optimization and Mixed Matrices

We now present a useful application for recursive matrices: Solving non-linear least squares problems. An iterative, local solution can be found by the Levenberg-Marquardt algorithm [Mor78]. In each step, a linear system of equations $H\Delta x = b$ is solved. The matrix H is the adjacency matrix of an undirected graph with parameters x_i as vertices and residuals r_i as edges [KGS*11]. Collapsing all vertices of the same type reveals the *type graph*. Fig. 1 shows the structure of pose graph optimization [GKSB10]. The type graph contains

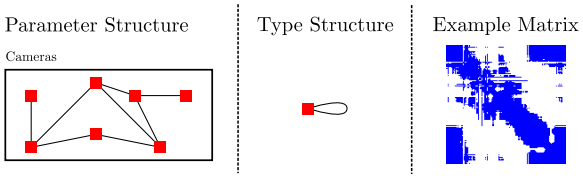

Figure 1: *Structure of pose graph optimization (SLAM).*

only a single node with a self-edge, because each residual depends on exactly two cameras. Since each pose has multiple parameters, a block structure arises. For example, given a 6 parameter rigid transformation, the recursive matrix type is:

```
using PGOMatrix =
  SparseSymmetric<Matrix<double, 6, 6>>;
```

For more complex optimization problems, the creation of a single recursive type is not possible with only conventional matrix types. Fig. 2 depicts the structure of bundle adjustment [TMHF99]. The resulting system matrix has a high level structure. The top left and bottom right are diagonal matrices and the rest is a general sparse matrix.

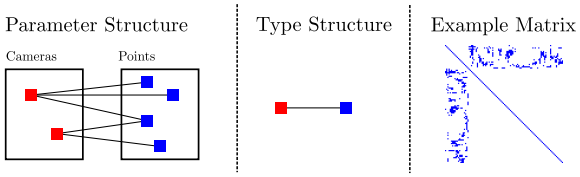

Figure 2: *Structure of bundle adjustment. Each edge connects one camera with one point. The type graph does not include self edges on the vertices.*

To this end, we define a new type of matrix which we will call *Mixed Matrix*. A mixed matrix is a generalized version of a dense matrix that allows each element to have a different type. Operations on mixed matrices behave identically to dense matrix operations. Using mixed matrices, we can now define H of any non-linear least squares problem recursively. For example, bundle adjustment with 6 camera and 3 point parameters results in the following type:

```
using BAMatrix = MixedMatrix22<
  DiagonalMatrix<Matrix<double, 6, 6>>,
  SparseMatrix<Matrix<double, 6, 3>>,
  SparseMatrix<Matrix<double, 3, 6>>,
  DiagonalMatrix<Matrix<double, 3, 3>>>;
```

These recursive mixed matrix types are derived from the type graph G of a given optimization problem. The root type is a dense mixed matrix R of size $n \times n$, where n is the number of different parameter types. If G contains a self edge on vertex i, then r_{ii} is a symmetric sparse matrix (see Fig. 1 as example). If G contains an edge between type vertex i and j, then R_{ij} and R_{ji} are general (non-symmetric) sparse matrices (see Fig. 2 as example). If a vertex i does not have a self edge, the corresponding diagonal element R_{ii} is a symmetric diagonal matrix. In all other cases, R_{ij} is zero. This is summarized in Equ. (5).

$$\text{Type}(R_{ij}) = \begin{cases} \mathit{SparseSymmetric} & \text{if } e_{i,j} \in G \wedge i = j \\ \mathit{Sparse} & \text{if } e_{i,j} \in G \wedge i \neq j \\ \mathit{DiagonalSymmetric} & \text{if } e_{i,j} \notin G \wedge i = j \\ \mathit{Zero} & \text{else} \end{cases} \tag{5}$$

5. Specialized Linear Solvers

In the previous section, we have shown how to construct the recursive mixed matrix H for non-linear least squares problems. The next step is to solve the corresponding linear system of equations. In popular optimization frameworks, for example Ceres and G2O, the user has to choose a linear solver that fits the problem. This leads to numerical errors or non-optimal solvers, if the user is not aware of the underlying algorithms. We observe that for many optimization problems, the Schur complement trick (Equ. (6)) increases the linear solver efficiency.

$$\begin{aligned} \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ (A - BD^{-1}C)x_1 &= b_1 - BD^{-1}b_2 \\ x_2 &= D^{-1}(b_2 - Cx_1) \end{aligned} \tag{6}$$

As seen in Equ. (6), the size of the linear system can be greatly reduced, but it also requires the inversion of one diagonal block. The Schur complement should therefore only be applied if the diagonal element is an *easily invertible* matrix. Given a recursively defined matrix type `T`, we define an inversion cost C_{-1} that describes how complex inverting a matrix of type `T` is (see Table 1). If this cost is below a threshold T_i for a diagonal element in H, the Schur complement trick improves the linear solver efficiency. For example, the system matrix of bundle adjustment (Fig. 2) consists of two block-diagonal regions. These block-diagonal matrices are easily invertible and therefore the Schur complement should be applied. If there are multiple viable Schur candidates, we choose the complement that produces the smaller reduced matrix $S = A - BD^{-1}C$. In the case of bundle adjustment with m points and n cameras, the point matrix is chosen for inversion, if $m \cdot 3 > n \cdot 6$.

Type	C_{-1}(Type)
`Real`	1
`DiagonalMatrix<T,n>`	$C_{-1}(T)$
`DenseMatrix<T,n,n>`	$C_{-1}(T) \cdot n^2$
`SymmetricMatrix<T,n,n>`	$C_{-1}(T) \cdot n^2 \cdot 0.5$
`MixedMatrix<T00,...,Tnn>`	$\max(C_{-1}(T_1),...) \cdot n^2$
`Matrix<T,Dynamic>`	∞
`SparseMatrix<T>`	∞

Table 1: *Inversion cost for different matrix types. This cost is evaluated at compile time and ∞ for all dynamically sized matrices.*

We have implemented such a linear solver for mixed matrices that analyzes the structure of H at compile time. The Schur complement is computed if applicable and the remaining linear system is solved using a direct or iterative solver. If there are multiple Schur candidates, code is generated for each possibility and the best is selected at run-time. As a direct solver, we use a recursive sparse Cholesky solver based on the Simplicial LDLT implementation of the Eigen library. The LDLT kernel was updated by applying the rules defined in Section 3. The iterative solver applies the preconditioned conjugate gradient algorithm. The diagonal preconditioner is computed recursively, which automatically results in a block Jacobi preconditioner for block-like system matrices.

6. Results

We have implemented the proposed types and algorithms as an extension to the C++ math library Eigen. The evaluation is split into two parts. First, we show that the recursively implemented basic linear algebra subroutines are on par or faster than state of the art BLAS libraries. Then, we compare our non-linear optimization

framework to other popular C++ frameworks and show that our system is on average 3-5 times faster. Both benchmarks are executed in double precision within a single thread on an Intel i7-7700K. The code is compiled by Clang 7.0 with all optimizations enabled.

6.1. Linear Algebra Subroutines

Currently, there are no other C++ math frameworks available that implement compile time recursive matrices. Therefore, we benchmark linear algebra operations only on level 2 recursive types, also known as block matrices. Intel's MKL library [Int03] is chosen as a reference, because it provides subroutines for sparse block matrices in BSR format. This format is binary compatible to a recursively defined sparse matrix of dense blocks.

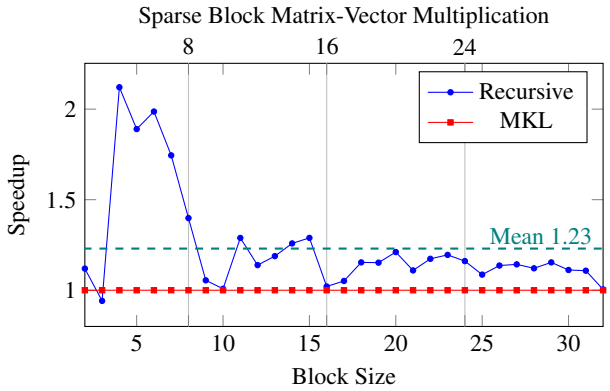

Figure 3: *Relative performance of the block-sparse matrix-vector product $y = A \cdot x + y$ at different block sizes. The matrix A is stored in BSR format.*

The performance of the block-sparse matrix-vector product is shown in Fig. 3. The results are normalized to visualize the relative speedup. On average, our implementation is 23.3% faster than MKL. At block sizes 16 and 32 the performance is roughly identical, which shows that Intel provides specialized kernels on par with our recursive implementation. The largest difference is achieved in the range 4 to 8 with a speedup of up to 2.12.

In Fig. 4, the performance of block-sparse matrix-matrix multiplication is shown. This benchmark does not include the optional column sorting step after the multiplication. The obtained speedup of our implementation increases with block size. If the block size is a multiple of 2 or 4, SSE and AVX instructions are generated by the compiler, which further increase the difference to MKL. The largest speedup is measured at a block size of 4 (= 5.15 times faster than MKL) and a block size of 8 (= 3.85 times faster than MKL).

6.2. Linear Solvers

To solve linear systems, we have implemented recursive versions of the PCG algorithm and the Cholesky factorization. The performance of PCG is not included in this section, because it is limited by the matrix vector product, which we have benchmarked in the previous section (Fig. 3).

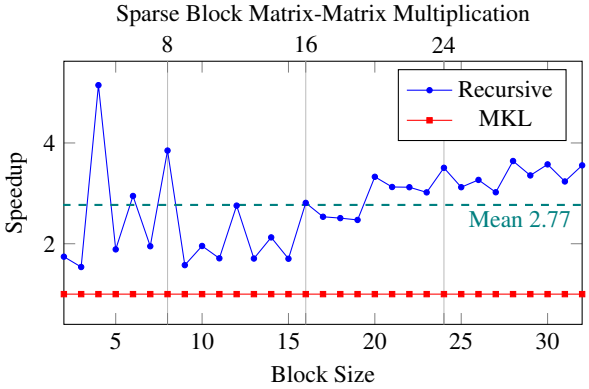

Figure 4: *Relative performance of the block-sparse matrix-matrix product $C = A \cdot B$ at different block sizes.*

The recursive Cholesky factorization is evaluated on different block sizes in Fig. 5. As a reference implementation, we use the Cholmod library [CDHR08], which is part of the SuiteSparse linear algebra package. Cholmod provides two different factorization algorithms: Simplicial and Supernodal. Simplicial is similar to Eigen's implementation, which is used as a base for the recursive algorithm. In Supernodal, the matrix structure is further analyzed to solve sub problems with dense matrix kernels (the BLAS [LHKK77]). This improves performance, if the matrix consists of large rectangular dense blocks, but is usually worse on very sparse or unstructured matrices. In Fig. 5 the relative performance between our recursive algorithm and Cholmod's Simplicial and Supernodal factorization is shown.

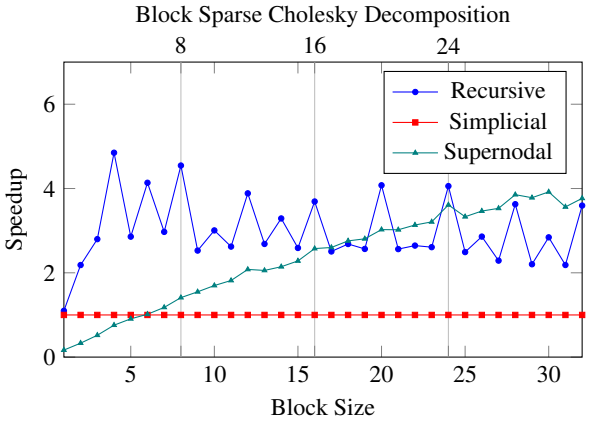

Figure 5: *Relative performance of the sparse Cholesky decomposition at different block sizes. The decomposed symmetric matrix contains 0.5% non zeros.*

The input matrix A has a Laplace-like pattern with five non-zero blocks per row. The total density of A is 0.5%. For all three factorization methods, the same AMD reordering scheme is used [ADD96]. The recursive and Simplicial algorithm compute the factorization $A = LDL^T$, where L is a lower triangular matrix with unit

Dataset	(1)		(2)		(3)		(4)		(5)	
Name	Trafalgar		Dubrovnik		SLAM		Venice		Ladybug	
# Cameras	138		356		772		1184		1723	
# Points	43699		225941		22147		805547		155481	
Density	0.363		0.536		0.120		0.278		0.080	
Solver Type	Iterative	Direct	Iterative	Direct	Iterative	Direct	Iterative	Direct	Iterative	Direct
Recursive	0.294	**0.217**	**2.404**	3.261	**0.276**	0.765	**9.514**	27.57	**1.253**	11.30
G2O	0.780	0.784	7.923	7.993	0.994	1.227	33.03	39.34	4.213	8.602
Ceres	1.278	1.345	10.27	14.50	1.613	2.087	24.59	62.10	5.596	10.537

Table 2: *Time in seconds for 5 iterations of bundle adjustment.*

diagonal and D is a diagonal matrix. The Supernodal algorithm, which only works for positive matrices, computes the factorization $A = LL^T$, with L being a lower triangular matrix. For small and medium block sizes (1-18), our recursive factorization outperforms both of Cholmod's algorithms. On large block sizes (21+) the Supernodal method requires the least amount of time. Because of vectorization, the achieved speedup of our method varies between odd and even block sizes. The largest difference is measured at 4×4 blocks, on which our recursive method is 4.8 times faster than Simplicial and 6.4 times faster than Supernodal.

6.3. Non-linear least squares optimization

Using the linear algebra subroutines and the linear solvers, we have implemented a small optimization framework for non-linear least squares problems. In this section, our framework is compared to Ceres [AMO12] and G2O [KGS*11] for three structured optimization problems of visual computing: Bundle Adjustment (BA) [TMHF99], Pose Graph Optimization [GKSB10], and ARAP [SA07]. A direct implementation of these problems with BLAS kernels is not possible, because the common libraries only support square inner blocks. In BA, a matrix multiplication with rectangular elements is required. Both Ceres and G2O use the Cholmod library whenever the solver type is set to direct. This also switches automatically between the Simplicial and Supernodal algorithm based on a sparsity heuristic. If the solver is set to iterative, the linear system is solved by the precondition conjugate gradient (PCG) method. The Schur complement is used in BA to reduce the complexity of the linear system.

Table 2 shows the average time in seconds for 5 iterations of BA. The preprocessing phase is executed once at the beginning and included in the benchmark. The performance is tested on four datasets (1,2,4,5) from the BAL benchmark [ASSS10] and one SLAM dataset (3) from the TUM RGB-D benchmark [SEE*12]. The iterative Schur complement solver of our recursive framework is 3-5 times faster than G2O and Ceres. This holds true for small dense as well as large sparse datasets. The difference is caused by multiple factors. First, G2O and Ceres require more preprocessing time to analyze the structure and select the appropriate types and solvers. In our framework, the structure is encoded in mixed matrices (see Section 4) and is therefore known at compile time. Ceres also uses sparse block matrices with dynamically sized blocks. This reduces the performance of all matrix operations, because the compiler cannot generate unrolled or vectorized code. When using a direct Cholesky solver, our recursive algorithm is faster for small and medium scenes. On large scenes (3)-(5), Cholmod's Supernodal factorization beats the recursive implementation. Our direct solver, which was also timed in Table 2, switches automatically to the Supernodal implementation based on a heuristic. For example, on the Ladybug dataset (5), our framework is around 6 times faster than G2O and Ceres, even though the same linear solver is used. In this case, the bottleneck is the Schur complement computation, which benefits from our highly efficient block-sparse matrix-matrix multiplication (see Section 6.1).

The performance of one iteration of pose graph optimization and ARAP (As Rigid As Possible) are shown in Table 3 and 4, respectively. In both cases, the PCG algorithm is used. The performance results are similar to bundle adjustment. Our recursive solver achieves a speedup of 3-5 in comparison to G2O and Ceres.

Dataset	(1)	(2)	(3)	(4)
Name	Trafalgar	SLAM	Ladybug	Venice
Recursive	**0.023**	**0.159**	**0.322**	**1.164**
G2O	0.134	1.025	2.167	7.511
Ceres	0.105	0.781	1.623	6.065

Table 3: *Time in seconds for 5 iterations of pose graph optimization using an iterative linear solver.*

Vertices	5k	12.5k	50k	125k
Density	$7 \cdot 10^{-4}$	$3.2 \cdot 10^{-4}$	$8 \cdot 10^{-5}$	$3.2 \cdot 10^{-5}$
Recursive	**0.249**	**0.722**	**3.798**	**10.46**
G2O	2.236	6.035	24.40	61.34
Ceres	1.195	3.197	13.33	33.67

Table 4: *Time in seconds for 5 iterations of ARAP using an iterative linear solver.*

7. Conclusion and Limitations

We have presented a linear algebra framework for recursive matrix operations. It is a header-only extension to the C++ math library Eigen and freely available at [Rü19]. We support all dense and sparse basic matrix operations for recursive types as well as a conjugate gradient solver and a sparse Cholesky decomposition. Additionally, we define dense mixed matrices, in which each element can be of a different type. A common use case is efficiently

solving structured non-linear least squares problems. The problem structure is fully encoded in the type, which reduces run-time overhead and allows advanced compiler optimizations.

We currently see two limitations. First, defining non-linear least squares problems in a recursive manner is more difficult than the run-time structure analyses performed by the Ceres solver. And second, due to our recursive types and the current implementation of Eigen's matrix operations, the quality of the generated code strongly depends on compiler optimizations. This results in large performance difference between current compilers. For example, the recursive bundle adjustment runs around 3 times slower when compiled with MSVC compared to Clang or GCC.

References

[ADD96] AMESTOY P. R., DAVIS T. A., DUFF I. S.: An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications 17*, 4 (1996), 886–905.

[AGK*00] ANDERSEN B. S., GUSTAVSON F., KARAIVANOV A., MARINOVA M., WAŚNIEWSKI J., YALAMOV P.: Lawra linear algebra with recursive algorithms. In *International Workshop on Applied Parallel Computing* (2000), Springer, pp. 38–51.

[AMO12] AGARWAL S., MIERLE K., OTHERS: Ceres solver. http://ceres-solver.org, 2012.

[AP86] AXELSSON O., POLMAN B.: On approximate factorization methods for block matrices suitable for vector and parallel processors. *Linear algebra and its applications 77* (1986), 3–26.

[ASSS10] AGARWAL S., SNAVELY N., SEITZ S. M., SZELISKI R.: Bundle adjustment in the large. In *European conference on computer vision* (2010), Springer, pp. 29–42.

[AWG01] ANDERSEN B. S., WAŚNIEWSKI J., GUSTAVSON F. G.: A recursive formulation of cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software (TOMS) 27*, 2 (2001), 214–244.

[Beb08] BEBENDORF M.: *Hierarchical matrices*. Springer, 2008.

[BG08] BULUC A., GILBERT J. R.: Challenges and advances in parallel sparse matrix-matrix multiplication. In *2008 37th International Conference on Parallel Processing* (2008), IEEE, pp. 503–510.

[BSL*16] BERNSTEIN G. L., SHAH C., LEMIRE C., DEVITO Z., FISHER M., LEVIS P., HANRAHAN P.: Ebb: A dsl for physical simulation on cpus and gpus. *ACM Transactions on Graphics (TOG) 35*, 2 (2016), 21.

[CDHR08] CHEN Y., DAVIS T. A., HAGER W. W., RAJAMANICKAM S.: Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS) 35*, 3 (2008), 22.

[DEF*13] DEMMEL J., ELIAHU D., FOX A., KAMIL S., LIPSHITZ B., SCHWARTZ O., SPILLINGER O.: Communication-optimal parallel recursive rectangular matrix multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (2013), IEEE, pp. 261–272.

[DMZ*17] DEVITO Z., MARA M., ZOLLHÖFER M., BERNSTEIN G., RAGAN-KELLEY J., THEOBALT C., HANRAHAN P., FISHER M., NIESSNER M.: Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *ACM Transactions on Graphics (TOG) 36*, 5 (2017), 171.

[EGJK04] ELMROTH E., GUSTAVSON F., JONSSON I., KÅGSTRÖM B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM review 46*, 1 (2004), 3–45.

[GHJ*98] GUSTAVSON F., HENRIKSSON A., JONSSON I., KÅGSTRÖM B., LING P.: Recursive blocked data formats and blas's for dense linear algebra algorithms. In *International Workshop on Applied Parallel Computing* (1998), Springer, pp. 195–206.

[GJ*10] GUENNEBAUD G., JACOB B., ET AL.: Eigen v3. http://eigen.tuxfamily.org, 2010.

[GKSB10] GRISETTI G., KUMMERLE R., STACHNISS C., BURGARD W.: A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine 2*, 4 (2010), 31–43.

[GVDG08] GOTO K., VAN DE GEIJN R.: High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw. 35*, 1 (2008), 4–1.

[HSHvdG16] HUANG J., SMITH T. M., HENRY G. M., VAN DE GEIJN R. A.: Strassen's algorithm reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2016), SC '16, IEEE Press, pp. 59:1–59:12. URL: http://dl.acm.org/citation.cfm?id=3014904.3014983.

[Int03] INTEL: Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/, 2003.

[KGS*11] KÜMMERLE R., GRISETTI G., STRASDAT H., KONOLIGE K., BURGARD W.: g 2 o: A general framework for graph optimization. In *2011 IEEE International Conference on Robotics and Automation* (2011), IEEE, pp. 3607–3613.

[KKRK*16] KJOLSTAD F., KAMIL S., RAGAN-KELLEY J., LEVIN D. I., SUEDA S., CHEN D., VOUGA E., KAUFMAN D. M., KANWAR G., MATUSIK W., ET AL.: Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG) 35*, 2 (2016), 20.

[LHKK77] LAWSON C. L., HANSON R. J., KINCAID D. R., KROGH F. T.: Basic linear algebra subprograms for fortran usage.

[Mor78] MORÉ J. J.: The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*. Springer, 1978, pp. 105–116.

[NP93] NG E. G., PEYTON B. W.: Block sparse cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing 14*, 5 (1993), 1034–1056.

[Rü19] RÜCKERT D.: Eigen recursive matrix extension. https://github.com/darglein/EigenRecursive, 2019.

[SA07] SORKINE O., ALEXA M.: As-rigid-as-possible surface modeling. In *Symposium on Geometry processing* (2007), vol. 4, pp. 109–116.

[SEE*12] STURM J., ENGELHARD N., ENDRES F., BURGARD W., CREMERS D.: A benchmark for the evaluation of rgb-d slam systems. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), IEEE, pp. 573–580.

[Str69] STRASSEN V.: Gaussian elimination is not optimal. *Numer. Math. 13*, 4 (Aug. 1969), 354–356. URL: http://dx.doi.org/10.1007/BF02165411, doi:10.1007/BF02165411.

[TMHF99] TRIGGS B., MCLAUCHLAN P. F., HARTLEY R. I., FITZGIBBON A. W.: Bundle adjustment—a modern synthesis. In *International workshop on vision algorithms* (1999), Springer, pp. 298–372.

[VM05] VUDUC R. W., MOON H.-J.: Fast sparse matrix-vector multiplication by exploiting variable block structure. In *International Conference on High Performance Computing and Communications* (2005), Springer, pp. 807–816.

[VZCvdG*09] VAN ZEE F., CHAN E., VAN DE GEIJN R., QUINTANA E., QUINTANA-ORTI G.: Introducing: The libflame library for dense matrix computations. *Computing in science & engineering* (2009).

[WOV*07] WILLIAMS S., OLIKER L., VUDUC R., SHALF J., YELICK K., DEMMEL J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (2007), IEEE, pp. 1–12.