



DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ACCELERATED SPARSE MATRIX OPERATIONS IN NONLINEAR LEAST SQUARES SOLVERS

AKCELERACE OPERACÍ NAD ŘÍDKÝMI MATICEMI
V NELINEÁRNÍ METODĚ NEJMENŠÍCH ČTVERCŮ

PH.D. THESIS
DISERTAČNÍ PRÁCE

AUTHOR
AUTOR PRÁCE

ING. LUKÁŠ POLOK

SUPERVISOR
VEDOUCÍ PRÁCE

DOC. RNDR. PAVEL SMRŽ, PH.D.

Let us approach this dedication vaguely chronologically ...

To my sister for eating my chemistry kit back when I was a kid,
to my parents for getting me all those computer parts,
to my advisor Pavel Smrž for the freedom he gave me,
to Vít'a Beran for involving me in his brand new robotics group,
to Viorela Ila who was a big source of inspiration and many interesting problems
and to my fantastic office mates who behaved like bewildered animals at all times.

ABSTRACT

This thesis focuses on data structures for sparse block matrices and the associated algorithms for performing linear algebra operations that I have developed. Sparse block matrices occur naturally in many key problems, such as Nonlinear Least Squares (NLS) on graphical models. NLS are used by e.g. Simultaneous Localization and Mapping (SLAM) in robotics, Bundle Adjustment (BA) or Structure from Motion (SfM) in computer vision. Sparse block matrices also occur when solving Finite Element Methods (FEMs) or Partial Differential Equations (PDEs) in physics simulations.

The majority of the existing state of the art sparse linear algebra implementations use *elementwise* sparse matrices and only a small fraction of them support sparse block matrices. This is perhaps due to the complexity of sparse block formats which reduces computational efficiency, unless the blocks are very large. Some of the more specialized solvers in robotics and computer vision use sparse block matrices internally to reduce sparse matrix assembly costs, but finally end up converting such representation to an elementwise sparse matrix for the linear solver.

Most of the existing sparse block matrix implementations focus only on a single operation, such as the matrix-vector product. The solution proposed in this thesis covers a broad range of functions: it includes efficient sparse block matrix assembly, matrix-vector and matrix-matrix products as well as triangular solving and Cholesky factorization. These operations can be used to construct both direct and iterative solvers as well as to compute eigenvalues. Highly efficient algorithms for both Central Processing Units (CPUs) and Graphics Processing Units (GPUs) are provided.

The proposed solution is integrated in SLAM ++, a nonlinear least squares solver focused on robotics and computer vision. It is evaluated on standard datasets where it proves to significantly outperform other similar state of the art implementations, without sacrificing generality or accuracy in any way.

KEYWORDS

Nonlinear least squares; numerical methods; sparse block matrix; general purpose computations on graphics processing units.

ABSTRAKT

Tato práce se zaměřuje na datové struktury pro reprezentaci řídkých blokových matic a s nimi spojených výpočetních algoritmů, jež jsem navrhl. Řídké blokové matice se vyskytují při řešení mnoha dílčích problémů jako například při řešení metody nejmenších čtverců. Nelineární metoda nejmenších čtverců (NLS) je často aplikována v robotice pro řešení problému lokalizace robota (SLAM) nebo v příbuzných úlohách 3D rekonstrukce v počítačovém vidění (BA), (SfM). Problémy konečných elementů (FEM) a parciálních diferenciálních rovnic (PDE) v oboru fyzikálních simulací můžou také mít blokovou strukturu.

Většina existujících implementací řídké lineární algebry používají řídké matice s granularitou jednotlivých elementů a jen několik málo podporuje řídké blokové matice. To může být způsobeno složitostí blokových formátů, jež snižuje rychlost výpočtů, pokud bloky nejsou dost velké. Některé ze specializovaných NLS optimalizátorů v robotice a počítačovém vidění používají blokové matice jako interní reprezentaci, aby snížily cenu sestavování řídkých matic, ale nakonec tuto reprezentaci převedou na elementovou řídkou matici pro implementaci k řešení systémů rovnic.

Existující implementace pro řídké blokové matice se většinou soustředí na jedinou operaci, často násobení matice vektorem. Řešení navržené v této disertaci pokrývá širší spektrum funkcí: implementovány jsou funkce pro efektivní sestavení řídké blokové matice, násobení matice vektorem nebo jinou maticí a nechybí ani řešení trojúhelníkových systémů nebo Choleského faktorizace. Tyto funkce mohou být snadno použity ke řešení systémů lineárních rovnic pomocí analytických nebo iterativních metod nebo k výpočtu vlastních čísel. Jsou zde popsány rychlé algoritmy pro hlavní procesor (CPU) i pro grafické akcelerátory (GPU).

Navrhované algoritmy jsou integrovány v knihovně SLAM ++, jež řeší problém nelineárních nejmenších čtverců se zaměřením na problémy v robotice a počítačovém vidění. Je provedeno vyhodnocení na standardních datasetech kde navrhované metody dosahují výrazně lepších výsledků než dosavadní metody popsané v literatuře – a to bez kompromisů v přesnosti či obecnosti řešení.

KLÍČOVÁ SLOVA

Nelineární metoda nejmenších čtverců; numerické metody; řídké blokové matice; obecné výpočty na jednotkách grafických akcelerátorů.

BIBLIOGRAPHIC CITATION

Ing. Lukáš Polok: *Accelerated Sparse Matrix Operations in Nonlinear Least Squares Solvers*, doctoral thesis Brno, Brno University of Technology, Faculty of Information Technology, 2016.

DECLARATION

I declare that this dissertation thesis is my original work and that I have written it under the guidance of Doc. RNDr. Pavel Smrž, Ph.D.. All sources and literature that I have used during my work on the thesis are correctly cited with complete reference to the respective sources.

Brno, 2016

Ing. Lukáš Polok, August 27, 2016

ACKNOWLEDGMENTS

Many thanks to my supervisor Pavel Smrž, who was being patient and supportive throughout my whole study and who got me on research projects where I was given free rein and could develop my ideas. I would also like to thank my colleagues at the Graph group, especially Viorela Ila, Pavel Zemčík, Adam Herout, Marek Šolony and Pavel Svoboda for helpful and motivational comments, proof-reading as well as lots of help with implementation and benchmarking.

Specifically, the algorithms described in the chapter on batch solving were greatly aided by inputs from Viorela Ila and also from Jean-Marie Codol, who shared the implementation of his least squares solver that helped me greatly to grasp the basics. The algebraic incremental Cholesky factorization update, covariance recovery using the recursive formula and using the incremental covariance update were all developed with a significant input from Viorela Ila. Marek Šolony helped greatly with writing the early papers and he is the author of the implementation of 3D observation models and the associated graph parsers in SLAM ++. Both Marek Šolony and Pavel Svoboda invested significant efforts into development and implementation of a computer vision front-end which yielded some of the datasets used for evaluating the algorithms based on Schur complement. A special thanks to Jeff Clifford and Simon Pabst from Double Negative Visual Effects for providing real production data and for allowing me to use some of it in publications. I hereby gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used in evaluations throughout this thesis.

Last but not least, I'd like to thank the guys who kept the department's coffee machine going. God knows I would not be able to do anything without it.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

- [BEK⁺15] Josep Blat, Alun Evans, Hansung Kim, Evren Imre, Lukáš Polok, Viorela Ila, Nikos Nikolaidis, Pavel Zemčík, Anastasios Tefas, Pavel Smrž, Adrian Hilton, and Ioannis Pitas. Big data analysis for media production. *Proc. IEEE*, PP(99):1–30, 2015.
- [IPŠ⁺15] Viorela Ila, Lukáš Polok, Marek Šolony, Pavel Smrž, and Pavel Zemčík. Fast covariance recovery in incremental nonlinear least square solvers. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 4636–4643, May 2015.
- [IPŠS17] Viorela Ila, Lukáš Polok, Marek Šolony, and Pavel Svoboda. SLAM+++—A highly efficient and temporally scalable incremental SLAM framework. *Intl. J. of Robotics Research*, Online First(0):1–21, 2017.
- [PIS13a] Lukáš Polok, Viorela Ila, and Pavel Smrž. Cache efficient implementation for block matrix operations. In *Proc. of the High Performance Computing Symp.*, pages 698–706. ACM, 2013.
- [PIŠ⁺13b] Lukáš Polok, Viorela Ila, Marek Šolony, Pavel Smrž, and Pavel Zemčík. Incremental block Cholesky factorization for nonlinear least squares in robotics. In *Robotics: Science and Systems (RSS)*, 2013.
- [PIS14] Lukáš Polok, Viorela Ila, and Pavel Smrž. Fast radix sort for sparse linear algebra on GPU. In *Proc. of the High Performance Computing Symp.* ACM, 2014.
- [PIS15] Lukáš Polok, Viorela Ila, and Pavel Smrž. Fast sparse matrix multiplication on GPU. In *Proc. of the High Performance Computing Symp.* ACM, 2015.
- [PIS16] Lukáš Polok, Viorela Ila, and Pavel Smrž. 3D reconstruction quality analysis and its acceleration on GPU clusters. In *Proc. of the European Signal Processing Conf. IEEE*, 2016.
- [PKP⁺15] Simon Pabst, Hansung Kim, Lukáš Polok, Viorela Ila, Ted Waine, Adrian Hilton, and Jeff Clifford. Jigsaw: multi-modal big data management in digital film production. In *ACM SIGGRAPH 2015 Posters*, page 50. ACM, 2015.

PUBLICATIONS

- [PŠI⁺13a] Lukáš Polok, Marek Šolony, Viorela Ila, Pavel Smrž, and Pavel Zemčík. Incremental Cholesky factorization for least squares problems in robotics. In *IFAC Symp. on Intelligent Autonomous Vehicles (IAV)*, volume 8, pages 172–178, 2013.
- [PŠI⁺13b] Lukáš Polok, Marek Šolony, Viorela Ila, Pavel Zemčík, and Pavel Smrž. Efficient implementation for block matrix operations for nonlinear least squares problems in robotic applications. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. IEEE, 2013.

CONTENTS

1	INTRODUCTION	1
1.1	Dense and Sparse Problems	2
1.2	Focus of the Thesis	3
1.3	Contributions	6
1.4	Structure of the Thesis	8
1.5	Math Notations	9
i	BACKGROUND	11
2	LEAST SQUARES METHODS AND THEIR APPLICATIONS	13
2.1	Nonlinear Least Squares	13
2.2	Simultaneous Localization and Mapping	20
2.3	Bundle Adjustment and Structure from Motion	21
2.4	Finite Element Methods	24
3	SPARSE MATRIX REPRESENTATIONS	27
3.1	Coordinate Format	27
3.2	Sparse Diagonal	28
3.3	Skyline	29
3.4	Ellpack-Itpack	29
3.5	Jagged Diagonal	30
3.6	Compressed Sparse Column or Row	31
3.7	Block Compressed Sparse Row	32
3.8	Variable Block Compressed Sparse Row	33
4	A BRIEF REVIEW OF EXISTING NLS SOLVERS	35
4.1	TORO	35
4.2	iSAM	36
4.3	HOG-Man	37
4.4	sSBA	37
4.5	SPA	38
4.6	g2o	38
4.7	iSAM2	39
4.8	Ceres	40
4.9	Part Summary	42
ii	SLAM ++ THE SPARSE BLOCK MATRIX SOLVER	45
5	SLAM ++ BLOCK MATRIX DESIGN	47
5.1	Related Work	49

5.2	Proposed Implementation	52
5.3	Performance Analysis	64
5.4	Chapter Summary	71
6	BATCH SOLVING IN SLAM ++	73
6.1	Related Work	74
6.2	Incremental SLAM	75
6.3	Implementation Details	76
6.4	Experimental Evaluation	77
6.5	Chapter Summary	82
7	INCREMENTAL SOLVING IN SLAM ++	85
7.1	Related Work	86
7.2	Incremental SLAM	87
7.3	Algebraic Incremental Updates of the Cholesky Factor	88
7.4	Implementation Details	90
7.5	Experimental Results	94
7.6	Improved Algorithm Using Block Cholesky Factorization	96
7.7	Incremental Updates of the Factor Using Resumed Cholesky	98
7.8	Experimental Evaluation	103
7.9	Chapter Summary	108
8	SOLVING BUNDLE ADJUSTMENT PROBLEMS	111
8.1	Finding Good Ordering	114
8.2	Incremental Solving	116
8.3	Nested Schur Complement	119
8.4	Experimental Evaluation	120
8.5	Chapter Summary	124
9	COVARIANCE RECOVERY	127
9.1	Related Work	128
9.2	Recovering Covariance in General NLS Problems	129
9.3	Recovering Covariance in Schur Complemented Systems	142
9.4	Chapter Summary	145
iii	ACCELERATING THE CHOSEN ALGORITHMS ON GPU	147
10	ACCELERATING THE CHOSEN ALGORITHMS ON GPU	149
10.1	A Brief History of GPU Computing	149
11	PARALLEL SORTING ON GPU	151
11.1	Related Work	152
11.2	Proposed Implementation	153
11.3	Performance Analysis	160
11.4	Chapter Summary	163

12	FAST SPARSE MATRIX MULTIPLICATION ON GPU	165
12.1	Related Work	166
12.2	Algorithm Design	167
12.3	Implementation	171
12.4	Results	175
12.5	Chapter Summary	178
13	ACCELERATING THE NONLINEAR LEAST SQUARES SOLVERS ON GPU	181
iv	CONCLUSIONS	185
14	CONCLUSIONS	187
14.1	Future Work	190
v	APPENDIX	193
A	APPENDIX	195
A.1	Typelist	195
A.2	Compiler-generated Decision Tree	196
	BIBLIOGRAPHY	201

LIST OF FIGURES

Figure 1.1	Examples of approximately block matrices.	4
Figure 1.2	An example of a randomly generated sparse block matrix. . .	5
Figure 1.3	Distribution of data between elementwise and block sparse matrices.	6
Figure 2.1	Example of pose and landmark SLAM	21
Figure 2.2	Graph of the <i>Venice</i> dataset.	22
Figure 2.3	Example of a uniform FEM mesh and the associated matrix.	25
Figure 3.1	Example sparse matrices.	27
Figure 3.2	Conversion of a matrix to the jagged diagonal format.	30
Figure 3.3	Example sparse block matrices.	32
Figure 5.1	Block placements in sparse block matrices.	47
Figure 5.2	Relation of expressions on block and elementwise sparse matrices.	48
Figure 5.3	Block row / column layout of a block matrix.	53
Figure 5.4	Performance scaling of compression of the MCCA matrix. . .	65
Figure 5.5	Performance scaling of general matrix vector product on the MCCA matrix.	66
Figure 5.6	Performance scaling of linear combination of the MCCA matrix and its transpose.	67
Figure 5.7	Performance scaling of the product of the MCCA matrix and its transpose.	68
Figure 5.8	The MCCA matrix and its split form.	70
Figure 5.9	Comparison of splitting and the variable block size approaches.	71
Figure 6.1	The synthetic datasets used in the batch solver evaluations.	76
Figure 6.2	The real-world datasets used in the batch solver evaluations.	76
Figure 6.3	Comparisons of time per vertex in the batch NLS solvers.	78
Figure 6.4	Comparisons of time per vertex in the batch NLS solvers running in incremental mode.	81
Figure 6.5	Time comparison of sparse block matrix operations performance on SLAM dataset matrices.	82

Figure 7.1	Evaluation of ordering heuristics in terms of nnz elements, compared to the Incremental R algorithm.	89
Figure 7.2	Fill-in relative to the AMD heuristic by elements.	90
Figure 7.3	Time comparison of multiple NLS optimizers.	95
Figure 7.4	Comparison of the χ^2 errors, on the 10k dataset.	96
Figure 7.5	Incremental updates on Λ -system and R-system.	98
Figure 7.6	Evaluation of ordering heuristics in terms of nnz elements, compared to the improved Incremental R algorithm.	99
Figure 7.7	Dataflow diagram of incremental block Cholesky factorization.	100
Figure 7.8	SLAM datasets used in the incremental solving evaluations.	104
Figure 7.9	Quality of the estimations measured on the 10k dataset.	107
Figure 7.10	Cholesky factorization benchmark on the standard SLAM datasets, a) times of factorization only and b) times of linear solving.	108
Figure 8.1	Sparsity patterns involved in common BA datasets in contrast to SLAM datasets.	112
Figure 8.2	Finding Schur ordering for landmark SLAM, on the Victoria Park dataset.	117
Figure 8.3	Examples of nested Schur complements using the AMICS ordering.	119
Figure 8.4	The BA datasets used in the Schur complement solving evaluations.	120
Figure 8.5	Evaluation of nested Schur orderings on standard SLAM and BA datasets.	122
Figure 8.6	Comparison of the effects of ordering on the linear solving on the standard BA datasets.	124
Figure 9.1	Distance-based candidates for data association calculated using the marginal covariances	128
Figure 9.2	Marginal covariances used as a quality estimate of 3D reconstruction.	128
Figure 9.3	Recovering the diagonal of the covariance matrix using the recursive formula.	130
Figure 9.4	Sparsity patterns involved in covariance update calculation.	131
Figure 9.5	Covariance recovery performance evaluation.	139
Figure 9.6	Covariance recovery memory usage benchmark.	141
Figure 9.7	Covariance precision on the Intel dataset.	142

LIST OF FIGURES

Figure 11.1	Relative amount of time spent in different phases of sparse matrix multiplication on GPU.	152
Figure 11.2	An example of segmented radix split operation for $b = 1$	158
Figure 11.3	GPU sorting rates on 32-bit keys.	162
Figure 11.4	GPU sorting rates on 32-bit key-value pairs.	162
Figure 11.5	GPU sorting rates on 32-bit key-value pairs, keys were generated in sparse matrix multiplication.	163
Figure 12.1	Time of different stages of the GPU matrix multiplication algorithm.	166
Figure 12.2	Data at the individual stages of the ESC algorithm.	168
Figure 12.3	Expansion factor by the number of product nonzero entries.	169
Figure 12.4	GPU matrix multiplication performance scaling comparison on Tesla K40.	176
Figure 12.5	Performance scaling comparison of sparse block matrix multiplication on the first four matrices from the SNAP dataset.	178
Figure 13.1	GPU-accelerated NLS solving performance on the standard BA datasets.	182

LIST OF TABLES

Table 2.1	A few of the commonly used robust functions.	18
Table 4.1	Overview of the state-of-the-art NLS solver data structures. .	42
Table 5.1	Timing results of sparse block matrix operations on a subset of University of Florida Sparse Matrix Collection.	69
Table 5.2	Timing results of sparse block Cholesky factorization on a subset of University of Florida Sparse Matrix Collection. . .	70
Table 6.1	Time comparisons of the batch NLS solvers.	78
Table 6.2	Time comparisons of the batch NLS solvers running in the incremental mode.	80
Table 7.1	Evaluation of incremental solving times of the NLS solvers. .	94
Table 7.2	Performance and accuracy tests of incremental NLS on the simulated datasets.	105
Table 7.3	Performance and accuracy tests of incremental NLS on the real-world datasets.	106
Table 8.1	Linear solving performance on the standard BA datasets. . .	121
Table 8.2	Incremental nonlinear solving performance on the standard BA datasets.	125
Table 9.1	Timing results of different state of the art covariance recov- ery implementations on multiple SLAM datasets.	138
Table 9.2	Characteristics of the BA datasets used in covariance recov- ery evaluations.	144
Table 9.3	Timing results and the associated space requirements of the evaluated covariance recovery methods.	145
Table 11.1	Memory complexities of segmented radix sort.	155
Table 11.2	Saturated GPU sorting performance.	161
Table 12.1	Matrices from the SNAP subset and CSpase matrix multi- plication performance.	176
Table 12.2	GPU sparse matrix multiplication performance comparison on the SNAP subset.	177

LIST OF TABLES

Table 13.1 GPU-accelerated NLS solving performance on the standard
BA datasets. 183

LIST OF LISTINGS

Listing 3.1	Example of a matrix stored in the C00 format.	28
Listing 3.2	Example of a matrix stored in the DIA format.	28
Listing 3.3	Example of a matrix stored in the SKY format.	29
Listing 3.4	Example of a matrix stored in the ELL format.	30
Listing 3.5	Example of a matrix stored in the JAD format.	31
Listing 3.6	Example of a matrix stored in the CSC format.	32
Listing 3.7	Example of a matrix stored in the BSR format.	33
Listing 3.8	Example of a matrix stored in the VBR format.	34
Listing 4.1	The g2o sparse block matrix format.	39
Listing 4.2	The Ceres sparse block matrix format.	41
Listing 5.1	The SLAM ++ sparse über block matrix format.	54
Listing A.1	A basic typelist.	195
Listing A.2	Storing values in a typelist.	195
Listing A.3	Decision tree skeleton.	197
Listing A.4	Wrapping an algorithm in a decision tree.	198
Listing A.5	Fixed block size matrix vector multiplication.	199
Listing A.6	Fixed block size matrix vector multiplication (continued). . .	200

LIST OF ALGORITHMS

Algorithm 3.1	A basic matrix - vector multiplication algorithm.	28
Algorithm 5.1	Naïve sparse block matrix multiplication.	57
Algorithm 5.2	Fast sparse block matrix multiplication.	58
Algorithm 5.3	Algorithm for Calculating Block Layout Mapping Function.	59
Algorithm 5.4	Two Dense Cholesky Factorization Algorithms.	62
Algorithm 7.1	Incremental SLAM algorithm using the algebraic R updates.	92
Algorithm 7.2	Gauss-Newton algorithm using the R factorization.	93
Algorithm 7.3	Incremental Block Cholesky Factorization.	101
Algorithm 7.4	Improved incremental SLAM algorithm.	103
Algorithm 8.1	Finding Maximum Independent Clique Sets.	115
Algorithm 9.1	Covariance recovery algorithm selection in an NLS solver	134
Algorithm 9.2	Blockwise covariance recovery.	135
Algorithm 9.3	Incremental covariance update.	136
Algorithm 11.1	Segmented parallel radix sort.	154
Algorithm 11.2	Naïve histogram calculation.	156
Algorithm 11.3	Registered histogram accumulation.	157
Algorithm 12.1	Setup stage of PSpGEMM.	171
Algorithm 12.2	Expansion and sorting stages of PSpGEMM.	172
Algorithm 12.3	Compression stage of PSpGEMM.	173

LIST OF ACRONYMS

GPU	Graphics Processing Unit
CPU	Central Processing Unit
GPGPU	general purpose computation on Graphics Processing Units
BLAS	Basic Linear Algebra Subprograms
LAPACK	Linear Algebra Package
SLAM	Simultaneous Localization and Mapping
SfM	Structure from Motion
BA	Bundle Adjustment
FEM	Finite Element Method
PDE	Partial Differential Equation
SIMD	Single Instruction Multiple Data
TLB	Translation Look-aside Buffer
LS	Least Squares
NLS	Nonlinear Least Squares
MIC	Many Integrated Cores
DOF	Degree of Freedom
MLE	Maximum Likelihood Estimation
BN	Bayes Net
MRF	Markov Random Field
FG	Factor Graph
LOT	Lapped Orthogonal Transform
IRLS	Iteratively Reweighted Least Squares
MAD	Median Absolute Deviation
KF	Kalman Filter

EKF	Extended Kalman Filter
UKF	Unscented Kalman Filter
IF	Information Filter
SAM	Smoothing and Mapping
RANSAC	Random Sample Consensus
GPS	Global Positioning System
RGBD	Red Green Blue Depth
IMU	Inertial Measurement Unit
GN	Gauss-Newton
LM	Levenberg-Marquardt
CG	Conjugate Gradient
FBS	Fixed Block Size
SSE	Streaming SIMD Extensions
FLOP	Floating Point Operation
FLOPS	Floating Point Operations per second
AMD	Approximate Minimum Degree
EMD	Exact Minimum Degree
MMD	Multiple Minimum Degree
RCM	Reverse Cuthill-McKee
MIS	Maximum Independent Set
MICS	Maximum Independent Clique Set
AMICS	Approximate Maximum Independent Clique Set
API	Application Programming Interface
CUDA	Compute Device Unified Architecture
OpenCL	Open Compute Library

INTRODUCTION

Many applications of numerical methods in many scientific disciplines can benefit from efficient implementations of linear algebra kernels. There are many implementations that provide comparable functionality, often providing standard Basic Linear Algebra Subprograms ([BLAS](#)) or Linear Algebra Package ([LAPACK](#)) interfaces that helped a great deal for linear algebra package development using a simple set of state-less C or Fortran functions. These functions are divided into several groups (or *levels*) by their complexity; L₁ contains the linear time functions on vectors, L₂ contains quadratic time matrix-vector functions and L₃ contains cubic time functions on matrices.

With the advent of C++, modern object-based interfaces with focus on intuitiveness, ease of use and safety became available. But that is not the only thing the object-based design has to offer: techniques such as expression templates can help fuse the computation kernels and reduce unnecessary data movement. The procedural and object-oriented approaches are not mutually exclusive: an efficient [BLAS](#) implementation can be conveniently wrapped in an expression templates interface.

Parallel implementations of [BLAS](#) kernels are the obvious next step to increase performance. Although the technologies are evolving constantly and Moore's law promises bigger Central Processing Units ([CPUs](#)) every year and a half, this no longer goes hand in hand with increasing clock frequencies. The era of constant increases in frequency and of architectural improvements that made newer [CPUs](#) faster "for free" is over. The performance is now obtained from parallelism, which requires effort also on the side of the algorithms and data structures.

While consumer multicore processors have been available since the early 2000s, the industry has not made major strides in the meantime – today's chips still have only up to 22 cores¹ in a single package. However, other architectures are available. One of those is the Graphics Processing Unit ([GPU](#)).

[GPUs](#) have been steadily gaining complexity for the past few years. Fueled by the massive entertainment industry, they provide relatively cheap performance. At first, they could only be utilized for computation by hacking the graphics pipeline. Later, specialized interfaces for general purpose computation on Graphics Processing Units ([GPGPU](#)) emerged that make it easier to leverage their performance for nongraphics applications, including linear algebra. [GPU](#) is a *streaming*-oriented ar-

Bigger, referring to a higher number of transistors – not faster! That would be a common misconception about Moore's law.

¹ E.g. a 22 core Xeon E5-2696 v4 released in April 2016, priced at \$4100.

architecture that focuses on raw processing power with thousands² of relatively simple cores organized in three tier hierarchy, with only a very small amount of cache available (hence streaming). The memory subsystem is also highly optimized as the memory resides directly on the GPU and cannot be changed or upgraded the way the CPU memory can.

Other architectures include e.g. Intel's Many Integrated Cores (MIC) architecture with hundreds³ of cores based on updated Pentium designs. Although the cores in different architectures are hardly comparable, this gives some idea about the levels of parallelism attainable on a modern workstation.

1.1 DENSE AND SPARSE PROBLEMS

Although seemingly very simple, the implementation of *dense* operations on modern hardware is not straightforward, if it needs to be done efficiently. This is due to the complexity of the CPUs in use today, which have a rather complex memory subsystem [48] with several levels of cache, support for paging and an autonomous prefetcher. There are also very fast Single Instruction Multiple Data (SIMD) instruction sets for arithmetics, with their own complicated rules.

To illustrate this with an example, a simple matrix product of the form $A \cdot B$ will run several times faster if A is first transposed, even at the cost of copying and reordering the data. To limit the amount of temporary storage and to otherwise aid the memory subsystem, dense routines are often *blocked*, meaning that the operation is not performed on the entire matrix at once but the matrix is divided into several blocks that are processed individually. High-performance implementations such as the Goto BLAS [69] focus on fine-tuning the sizes of blocks to match various machine limits (in this case the size of the Translation Look-aside Buffer (TLB)).

For certain applications, the matrices have a substantial portion of zero entries. Using dense matrix algorithms would be a waste of both memory and computation – that is where the *sparse* linear algebra comes in (and of course also sparse BLAS). For sparse algorithms, the matrix is represented in such a way that only the non-zero entries are stored and the computation can be performed efficiently both in terms of storage and the ratio of the arithmetic operations to the rest of the algorithm. Sparse algorithms are typically much more complicated compared with the dense algorithms, due to the necessity of matching the non-zero entries that interact in the given operation and at the same time forming the sparse structure in case the result is a matrix. Efficient sparse algorithms are usually a fine mix of numerical methods and graph theory. There is a certain threshold of *useful sparsity*

² E.g. NVIDIA Titan X introduced in May 2015 has 3072 cores and sells for about \$1500.

³ E.g. Xeon Phi 7120A released in April 2014 with 61 cores costs about \$4000.

beyond which it is better to just represent the matrix as a dense matrix, from the performance point of view.

To illustrate the difficulty in implementing efficient sparse operations, e.g. sparse matrix-vector multiplication algorithms often run at one tenth of the peak hardware performance [173] and the situation can be even worse for the matrix-matrix multiplication [16, 38]. This is due to irregularity of memory accesses and various other overheads. At the same time, those algorithms are typically much harder to adapt for hardware acceleration.

1.2 FOCUS OF THE THESIS

The general objective of this thesis is to identify a suitable class of problems and to propose a computation acceleration scheme. However, the topic of application of GPGPU to accelerate linear algebra is too wide to specify a clear research goal. Rather than pursuing fast implementations of a few randomly chosen algorithms, this thesis examines a particular class of applications that are commonly solved using numerical sparse linear algebra.

Several estimation problems fall into this category. In general, an estimation problem finds an optimal configuration of a set of variables given a vector of their initial values and a set of relations between those variables. If represented using a graph, the nodes in the graph are given by the variables to be estimated and the edges are the relations between those variables.

It is common to use tools such as graphical models to capture the structure and dependencies of the estimation problems. Bayes Nets (BNs), Markov Random Fields (MRFs) or Factor Graphs (FGs) are commonly used for this purpose. While BNs are linked to the generative aspects and explicitly show the dependencies of the variables in solving the problem, MRFs and FGs better capture the structure and the connection with the underlying linear algebra, in particular the matrices.

A condition for the problem to be sparse is that each of the variables must only relate to a small subset of the other variables. This translates into an underlying graph with a low maximum degree.

Examples of such problems can be found in robotics and computer vision. Simultaneous Localization and Mapping (SLAM) estimates the pose of a robot in conjunction with the map of the environment from various sensor measurements. Similarly, Bundle Adjustment (BA) or Structure from Motion (SfM) in computer vision estimate the camera parameters together with the 3D structure observed from different locations of the same or different cameras.

These problems have been widely studied in the past decades, yet the computational complexity is still an open issue. A SLAM problem in general grows with

This thesis will refer to FG for representing the estimation problems. FG is a bipartite graph where both the relations and the variables in the estimation are vertices.

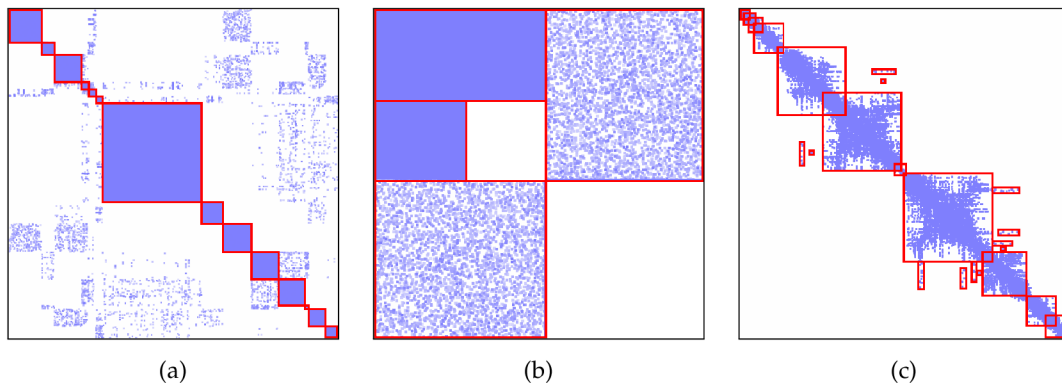


Figure 1.1: Examples of approximately block matrices from the University of Florida Sparse Matrix Collection [39], specifically in the DIMACS10 dataset [147], a) an approximate block matrix with scattered nonzero elements, b) a block matrix with unaligned blocks, and in the Oberwolfach dataset [110], c) an approximate block matrix with overlapping blocks. Note that the block boundaries (in red) are only suggested – not a part of the original matrices.

every step the robot takes, and for long runs (several days of robot operation) this can become intractable using limited computational resources on board a robotic platform. Similarly, reconstructing a large 3D environment using a BA algorithm may involve millions of variables.

To handle the inherent sensor noise, those problems are formulated in a probabilistic framework. Maximum Likelihood Estimation (MLE) is a way to incorporate noise models into the estimation problem. In general, those models are nonlinear (e.g. the motion model of a robot involves rotations, vision problems work with 3D projective geometry). Under the assumption of Gaussian noise, MLE has an elegant Nonlinear Least Squares (NLS) solution.

NLS problems are typically solved numerically, and that requires calculating derivatives to linearize the problem locally and then solve the resulting system of linear equations. In the above problems, each of the variables only has a limited number of relations to the others. In consequence, the Jacobian matrices obtained by calculating derivatives of the functions relating the estimated variables are sparse. Furthermore, those Jacobian matrices have a direct connection to the incidence matrix of the underlying graph. Similarly, the adjacency matrix corresponds to the Hessian matrices.

Another important characteristic of such problems is the fact that the variables are often multivariate, e.g. a 3D robot pose may have six Degrees of Freedom (DOFs) (three for position and three to represent the orientation), a landmark three DOFs. This structure appears implicitly in the resulting system matrices, where the elements corresponding to each variable can be conceptually grouped into blocks, giving rise to sparse *block* matrices.

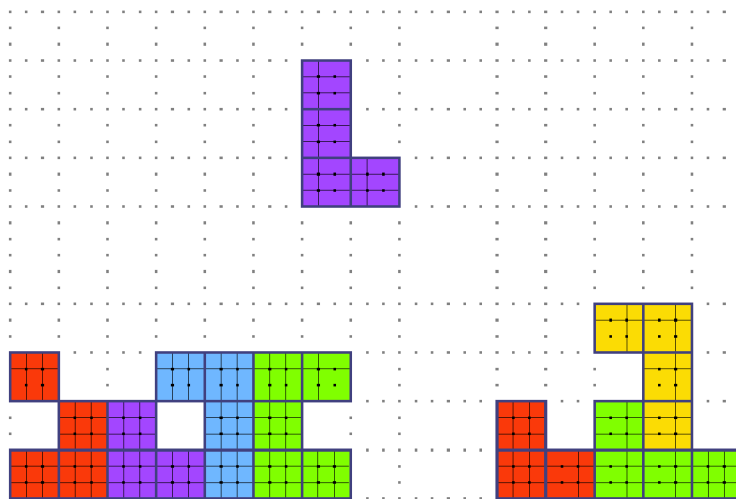


Figure 1.2: An example of a randomly generated sparse block matrix composed of 31 blocks, 3×3 elements each, used in testing operations on block matrices.

A block matrix is a matrix that is conceptually partitioned into blocks. A block matrix can have either an exact block pattern or an approximate one where scattered nonzero entries are allowed, as in Figure 1.1a. Another distinction is the presence of unaligned or overlapping blocks – whether the conceptual edges of a block could intersect those of another block, as in Figure 1.1c.

While approximate block patterns are sometimes employed to limit the required communication bandwidth in parallel algorithms [140, 164], this work relates to exact block patterns such as in the matrix in Figure 1.2. While one may object that such matrices are rare, the opposite is true. In Figure 1.3, there is a plot of the distribution of matrix nonzeros between elementwise and block matrices in the University of Florida Sparse Matrix Collection [39]. To generate it, the algorithm from [146] was employed to discover block structure in the matrices. The horizontal axis of the plot is given by the percentage of nonzeros of each given matrix residing in blocks of at least three elements. Although the *number* of block matrices is somewhat lower than that of sparse matrices, this plot shows that the majority of the *data* in this dataset is in fact in block matrices.

The focus of this thesis is to propose new algorithms and implementations to accelerate linear algebra operations in NLS problems with a sparse, block structure. A new data structure is proposed to benefit highly from the block structure and incremental nature of those problems, when iteratively calculating the solution of an NLS. Furthermore, the possibilities of GPU acceleration are explored. The thesis shows that the proposed methods supersede all existing implementations in this direction and generate state of the art algorithms for problems such as SLAM and BA or SfM.

The proposed solutions can also benefit other fields. In addition to the estimation problems described here, there are other problems with inherent block structure,

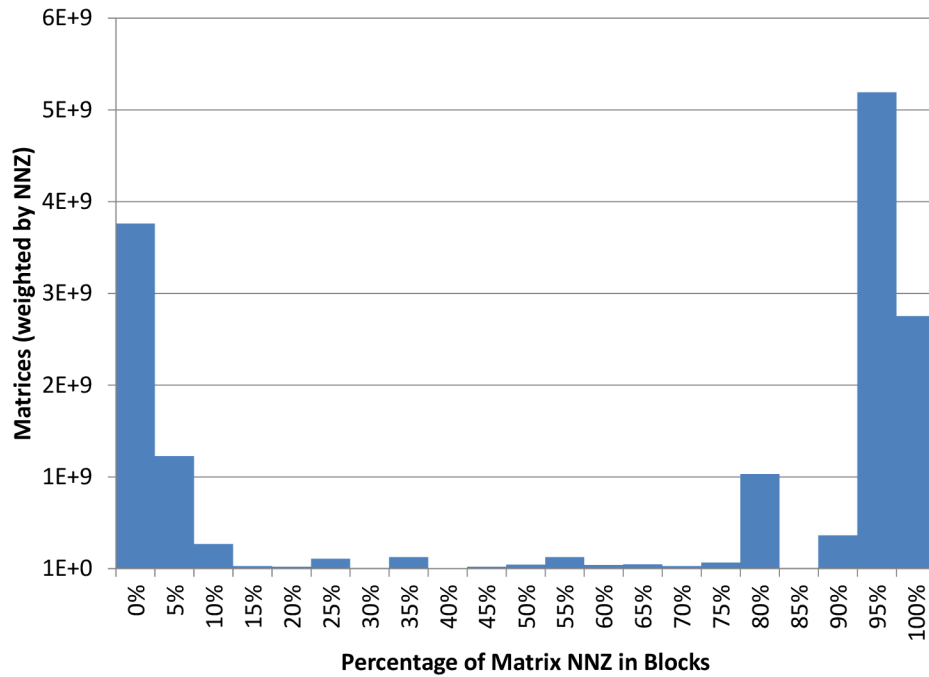


Figure 1.3: Distribution of data between elementwise and block sparse matrices in the University of Florida Sparse Matrix Collection [39].

such as Finite Element Methods (FEMs) or Partial Differential Equations (PDEs) in physics simulations which also have an underlying graph and a block structure, Lapped Orthogonal Transforms (LOTs) in image processing have a particular block structure. In addition, a number of methods exist [50, 51, 140, 90, 173, 175] to consolidate general sparse matrices into block matrices, making acceleration of problems without inherent block structure also possible.

1.3 CONTRIBUTIONS

The main contributions of my work described in this thesis are the following:

NOVEL DATA STRUCTURE FOR SPARSE BLOCK MATRICES: In this dissertation, an analysis of sparse matrix operations useful in NLS solving is presented. Based on this analysis, a novel data structure for sparse block matrices was designed. This also required implementation of efficient operations on those new matrices.

A NEW APPROACH TO LOOP UNROLLING: The arithmetic operations on the proposed sparse block matrices are optimized by loop unrolling (sometimes also referred to as *register blocking*). This was done using novel C++ constructs, based on BLAS kernel specialization using information about the input matrices that is available at compile-time. This block matrix implementation is

one of a very few implementations that support matrix factorizations also on matrices with multiple block sizes.

FAST NLS SOLVER BASED ON BLOCK MATRIX SCHEMES: The proposed block matrix scheme was demonstrated in a realistic scenario of NLS solving. Even without any algorithmic improvements on the solver part, the block matrix operations already give a significant performance advantage over the state of the art solvers while operating in batch mode.

FAST ALGORITHMS FOR INCREMENTAL SOLVING: Incremental solving was also investigated, and although the proposed block scheme offers some advantages, such as fast matrix modification when new constraints need to be integrated into the system, a basic Gauss-Newton solver cannot compete with the state of the art incremental solvers, despite being implemented efficiently. Two new methods for incremental solving are proposed, an algebraic method, which only takes advantage of elementwise sparse factorizations and a method taking advantage of the block approach, based on a novel algorithm called *resumed* Cholesky factorization and the corresponding algorithm for incremental variable reordering which keeps the incremented factor sparse.

NEW ALGORITHMS FOR SCHUR COMPLEMENT: A novel variable ordering based on cliques in the underlying graph was proposed, which yields some sort of a *supernodal* Schur complement. It offers advantages when solving with a dense linear solver (e.g. on a GPU). It can provide significant memory savings by reducing the size of the dense part, as well as promoting parallelism and cooperative CPU-GPU processing in inverting the block diagonal parts.

Incremental Schur complement equations were derived and benchmarked, yielding notable speedups and at the same time requiring modest amounts of memory.

FAST METHOD OF COVARIANCE RECOVERY: While estimating the mean of the observed variables is the central role of MLE, estimating the *covariances* can be equally important for some applications, yet it is often neglected by the state of the art implementations. Blockwise formulation of covariance recovery alone yields significant speedups compared with the state of the art.

A novel method for incremental covariance update was also proposed, yielding up to two orders of magnitude speedups and thus offering covariances at a cost comparable to that of a direct solver. The precision of incrementally calculated covariances is on a par with batch methods. An elegant variant of

Batch refers to solving the entire problem at once, while incremental refers to solving only a small part of the problem initially and then adding more variables and constraints and solving again, in incremental fashion.

update by downdate further reduces storage requirements in practical solver implementations while not sacrificing any of the performance.

FAST COVARIANCE RECOVERY FOR SCHUR-COMPLEMENTED SYSTEMS:

Complexities of covariance recovery in Schur-complemented systems are also investigated and efficient methods are proposed to recover the covariances of both the variables in the diagonal part and in the complement, observing up to an order of magnitude speedups when compared with the blockwise recursive formula.

Finally, the selected operations were implemented to run on a GPU, which is novel because block matrices were not widely attempted on GPUs. In the development of these implementations, improved algorithms for parallel sorting and sparse matrix-matrix multiplication on a GPU were developed.

All of the contributions (and more) form the basis of SLAM ++, a high-performance NLS solver based on sparse block matrices, focused especially on efficient incremental estimation (hence the ++, which means increment in the family of C languages). SLAM ++ is freely available under the MIT license and has been downloaded more than 3500 times from its website <http://sf.net/p/slam-plus-plus/> so far.

1.4 STRUCTURE OF THE THESIS

The next chapter serves a brief introduction into the NLS problems and their applications, along with their characteristics. The applications discussed have sparse structure and Chapter 3 gives an overview of commonly used and also some relevant but less used sparse matrix formats. Chapter 4 describes the state of the art NLS solver packages while also focusing on the matrix representation and numerical algorithms.

In the following part of the thesis, Chapter 5 describes the proposed novel sparse block matrix storage and the algorithms for performing allocation as well as numerical operations with matrices in this format. Chapter 6 describes the use of this new format in a batch NLS solver for SLAM problems. Real online problems in robotics, among others, require incremental solving which is described in Chapter 7. Some classes of problems, e.g. BA and SfM in computer vision, can be solved more efficiently using Schur complement, which is the topic of Chapter 8. Chapter 9 shows how covariance of the variables can be estimated, in addition to the mean, and briefly summarizes what are the uses for such covariances.

In the final part which opens with Chapter 10, the sole focus is on the acceleration of the algorithms on the GPU. Specifically, Chapter 11 describes efficient

sorting algorithm for GPUs and Chapter 12 describes fast sparse matrix multiplication. Those two elementary kernels allow acceleration of the algorithms proposed in the second part by removing their main bottlenecks and the results are summarized in Chapter 13.

The thesis concludes with Chapter 14. The APPENDIX contains some implementation details of the block format proposed in the second part.

1.5 MATH NOTATIONS

This text makes use of various more or less standard mathematical notations. This section briefly revises the used conventions. Vectors are denoted by small bold letters, e.g. v is a scalar but \mathbf{v} is a vector. Matrices are denoted by capital Latin or Greek letters, e.g. A and Λ are matrices. Matrices (and as a special case also vectors) which are logically partitioned to blocks are denoted by *bold* capital Latin or Greek letters, e.g. Σ is a matrix but $\mathbf{\Sigma}$ is a matrix where the elements are matrices.

To assemble a (column) vector, one writes $\mathbf{c} = [1; 2; 3]$ while to assemble a row vector, one writes $\mathbf{r} = [1, 2, 3]$. By an extension, $A = [1, 2; 3, 4]$ is the same as:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

Similarly, a block vector can be initialized as $\mathbf{J} = [\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}}, \frac{\partial \mathbf{r}}{\partial \boldsymbol{\iota}}, \frac{\partial \mathbf{r}}{\partial \boldsymbol{\kappa}}]$ where the expressions for the elements yield matrices (since \mathbf{r} , $\boldsymbol{\theta}$, $\boldsymbol{\iota}$ and $\boldsymbol{\kappa}$ are all vectors), or more expressively as $\mathbf{J} = [A, B, C]$ (where A , B and C are clearly matrices).

In some cases, a need arises to access elements of the matrices and vectors. To access an element of a vector, a subscript is used, e.g. \mathbf{v}_i is the i^{th} element of this vector. In some cases, this is similar to a scalar with a subscript in which case the v would be typeset in regular. Cases of a vector with a subscript which is not an index should be clear from the context (but the text mostly alerts the reader when that occurs). To select a *range* of elements of a vector, Matlab notation is used, so that $\mathbf{v}_{i:j}$ refers to a vector formed by concatenating $[\mathbf{v}_k \mid \forall k : i \leq k \leq j]$. For convenience, $\mathbf{v}_{i:\text{end}}$ refers to a vector formed by concatenating $[\mathbf{v}_k \mid \forall k : i \leq k]$.

Similarly, to get an *element* of a matrix, it is possible to use $A_{i,j}$ which selects an element at row i and column j (note the use of the comma separating i and j). To select an entire row of a matrix, it is possible to use $A_{i,*}$ where the asterisk reads as “any column”. The same is also possible for columns, e.g. $A_{*,i}$. It is possible to select ranges of elements, the same way as in vectors: $A_{i:j,k:l}$ selects a rectangular region of rows i through j and columns k to l . Similarly, $A_{i:\text{end},k:\text{end}}$ selects the bottom right corner of the matrix. Combinations with asterisk are also permitted, e.g. $A_{i:j,*}$ selects a range of rows.

In some cases, the matrices are partitioned to logical sections (but other than the blocks referred to before), for example a matrix Λ can be seen as:

$$\Lambda = \begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{pmatrix},$$

where the Λ_{ij} refers to a logical section of the matrix. Note that there is no comma separating the indices. At the same time $\Lambda_{k,l}$ refers to an element at row i and column j (or a block element if Λ were bold). For some $\Lambda_{k,l}$, it generally cannot be determined whether the element is e.g. in Λ_{11} or any other section unless the sizes of the sections are specified. This notation is used almost exclusively with numbers for the indices (rather than variables). This notation is sometimes also used for vectors where it is again aliasing with the element access but the meaning is made clear in such instances.

As far as indices are concerned, the algorithms and listings use zero-based indexing since that is more natural and usually leads to shorter index expressions. On the other hand, most of the math formulas will use one-based indexing as that is the de-facto norm. Indexing operations take precedence before matrix operations denoted by the superscript, such as transpose or inverse, so e.g. $\Lambda_{21}^\top = (\Lambda_{21})^\top \neq (\Lambda^\top)_{21}$.

In the description of the incremental solvers, there are instances of a variable from the previous time frame and from the next time frame coexisting together. This is denoted using the hat symbol, e.g. after an update, A becomes \hat{A} .

In the description of linear solving techniques, linear solving is denoted using the backslash operator, e.g. for a linear system $\Lambda x = \mathbf{b}$ with x being the unknown, solving is denoted $x = \Lambda \setminus \mathbf{b}$. This is especially applied to triangular systems which can be solved by backsubstitution, but the operator is general and can be applied to any system. The same operator is sometimes used in literature to denote Schur complements – but not in this thesis, in order to avoid confusion.

Part I

BACKGROUND

This part describes methods for nonlinear optimization and their applications, which in part serve as the motivation for proposing the accelerated data structures and algorithms in the latter parts of this thesis.

LEAST SQUARES METHODS AND THEIR APPLICATIONS

The following chapters describe nonlinear least squares on graphical models and approaches to finding their solutions efficiently. Practical problems which are usually solved using least squares are discussed and their particularities are pointed out. As the graph structure is usually quite sparse, it is suitable to represent it using a sparse matrix. The commonly used sparse matrix representations are listed, along with their advantages. Finally, state of the art least squares solvers are described, along with their own novel solutions to matrix representations and solving. This chapter comprises the foundation for the requirements from the linear algebra point of view.

The term “graphical model” refers to a configuration where the estimated variables comprise vertices of a graph, with the constraints being the edges; typically such graphs are far from being fully connected.

2.1 NONLINEAR LEAST SQUARES

Probabilistic methods have been extensively applied in robotics and computer vision to handle noisy perception of the environment and the inherent uncertainty in the estimation. There is a variety of solutions to the estimation problems in today’s literature. Filtering and Maximum Likelihood Estimation (MLE) are among the most used in robotics. Since filtering easily becomes inconsistent when applied to nonlinear processes [160], MLE gained a prime role among the estimation solutions. In Simultaneous Localization and Mapping (SLAM) [45, 95, 106, 98] or other mathematically equivalent problems such as Bundle Adjustment (BA) [4, 105] or Structure from Motion (SfM) [14], the estimation problem is solved by finding the MLE of a set of variables (e.g. camera/robot poses and 3D points in the environment) given a set of observations. Assuming Gaussian noises and processes, the MLE has an elegant Nonlinear Least Squares (NLS) solution.

In practice, the initial problem is nonlinear and it is usually addressed by repeatedly solving a sequence of linear systems. The linear system can be solved either by matrix factorization or gradient methods. The latter are more efficient from the storage point of view, since they only require access to the gradient, but they can suffer from poor convergence, slowing down the execution. Matrix factorization, on the other hand, produces more accurate solutions and avoids convergence difficulties but typically requires a lot of storage.

In this context, the estimation problem is formulated as a maximum likelihood estimation of a set of variables $\theta = [\theta_1 \dots \theta_n]$ given a set of observations

In here, $\mathbf{z} = [z_1 \dots z_m]$. Two basic modes of operation can be distinguished. The *batch* operation consists of obtaining a bulk of initial values of the variables and a bulk of measurements which specifies the task to be solved and solving it until convergence. This is useful especially in the offline applications. On the other hand, the estimation has to be done *incrementally* in an online application; at every step new variables and the associated measurements are integrated into the system and a new solution is calculated.

In this section, we briefly show how the **MLE** problem is formulated and solved using **NLS**. The joint probability distribution can be written as:

$$P(\boldsymbol{\theta}, \mathbf{z}) \propto P(\boldsymbol{\theta}_0) \prod_z^n P(z_k | \theta_{i_k}, \theta_{j_k}), \quad (2.1)$$

where $P(\boldsymbol{\theta}_0)$ is the prior and z_k are the constraints between the variables θ_{i_k} and θ_{j_k} . The goal is to obtain the **MLE** of a set of variables in $\boldsymbol{\theta}$, given the available observations in \mathbf{z} :

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} P(\boldsymbol{\theta} | \mathbf{z}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} (-\log(P(\boldsymbol{\theta} | \mathbf{z}))). \quad (2.2)$$

For simplicity of the notation, the constraints in this formulation are binary. Unary or hyper-constraints are also sometimes needed.

For every measurement $z_k = h_k(\theta_{i_k}, \theta_{j_k}) - v_k$, the noise v_k is assumed to be normally distributed, with zero mean and covariance Σ_k :

$$P(z_k | \theta_{i_k}, \theta_{j_k}) \propto \exp\left(-\frac{1}{2} \|h_k(\theta_{i_k}, \theta_{j_k}) \ominus z_k\|_{\Sigma_k}^2\right), \quad (2.3)$$

where $h_k(\theta_{i_k}, \theta_{j_k})$ is the nonlinear measurement function, z_k are the measurements, \ominus is the vectorial inverse composition operator. Note that binary measurements are assumed here but measurements of any degree can be combined at will. Setting $\boldsymbol{\Sigma} = \mathbf{I}$ yields ordinary nonlinear least squares, otherwise weighted **NLS** are obtained. Finding the **MLE** from (2.2) is done by solving the following **NLS** problem:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(\frac{1}{2} \sum_{k=1}^m \|h_k(\theta_{i_k}, \theta_{j_k}) \ominus z_k\|_{\Sigma_k}^2 \right). \quad (2.4)$$

Gathering all residuals in $\mathbf{r}(\boldsymbol{\theta}) = [r_1, \dots, r_m]^T$ where $r_k = h_k(\theta_{i_k}, \theta_{j_k}) \ominus z_k$ and gathering the measurement noise in $\boldsymbol{\Sigma} = \operatorname{diag}([\Sigma_1, \dots, \Sigma_m])$, the sum in (2.4) can be written in the vectorial form and expressed in terms of a L2-norm:

$$\|\mathbf{r}(\boldsymbol{\theta})\|_{\boldsymbol{\Sigma}}^2 = \mathbf{r}^T(\boldsymbol{\theta}) \boldsymbol{\Sigma}^{-1} \mathbf{r}(\boldsymbol{\theta}) = \|\boldsymbol{\Sigma}^{-T/2} \mathbf{r}(\boldsymbol{\theta})\|^2. \quad (2.5)$$

Iterative methods, such as Gauss-Newton are often used to solve the **NLS** in (2.4). This is usually addressed by repeatedly linearizing the problem, solving the obtained linear system and updating the estimate. Linear approximations of the nonlinear residual functions around the current estimate $\boldsymbol{\theta}^i$ are calculated as:

$$\hat{\mathbf{r}}(\boldsymbol{\theta}^i) = \mathbf{r}(\boldsymbol{\theta}^i) + \mathbf{J}(\boldsymbol{\theta}^i)(\boldsymbol{\theta} \ominus \boldsymbol{\theta}^i), \quad (2.6)$$

with J being the Jacobian matrix which gathers the derivatives of the components of $\mathbf{r}(\boldsymbol{\theta})$. With this, the NLS in (2.4) is approximated by a linear one and solved by successive iterations:

$$\boldsymbol{\delta}^* = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \frac{1}{2} \|\Lambda \boldsymbol{\delta} - \mathbf{b}\|^2, \quad (2.7)$$

where the matrix A and the vector \mathbf{b} are defined [45] as:

$$A \triangleq \boldsymbol{\Sigma}^{-1/2} J \quad (2.8)$$

$$\mathbf{b} \triangleq -\boldsymbol{\Sigma}^{-1/2} \mathbf{r}. \quad (2.9)$$

The correction $\boldsymbol{\delta} \triangleq \boldsymbol{\theta} \ominus \boldsymbol{\theta}^i$ towards the solution is obtained by solving the linear system:

$$A^\top A \boldsymbol{\delta} = A^\top \mathbf{b}, \quad (2.10)$$

where we define the *information matrix* $\Lambda \triangleq A^\top A$ and the right hand side (r.h.s) $\boldsymbol{\eta} \triangleq A^\top \mathbf{b}$. The linear system becomes:

$$\Lambda \boldsymbol{\delta} = \boldsymbol{\eta}, \quad (2.11)$$

and is commonly referred to as the *normal equation*. The particular state of $\boldsymbol{\theta}^i$ for which the derivatives in J are computed is referred to as the *linearization point*. In order to obtain the solution of this linear system, it is common to apply matrix factorization, followed by back-substitution. The Cholesky factorization of the matrix Λ has the form $R^\top R = \Lambda$, where R is an upper triangular matrix with positive diagonal entries. The forward and back-substitutions on $R^\top \mathbf{d} = \boldsymbol{\eta}$ and $R \boldsymbol{\delta} = \mathbf{d}$ first recover \mathbf{d} and then the actual solution $\boldsymbol{\delta}$. Alternatives to the Cholesky factorization in the form of RDR^\top or LU decomposition do not offer great advantages in solving the normal equation while being slightly slower.

Alternatively, the normal equation in (2.11) can be skipped and *QR factorization* can be applied directly to the matrix A in (2.7), yielding $A = QR$. The solution $\boldsymbol{\delta}$ can be directly obtained by back-substitution in $R\boldsymbol{\delta} = Q^{-1}\boldsymbol{\eta}$ where $Q^{-1} = Q^\top$ as Q is orthogonal. Note, that Q is not explicitly formed; it is commonly represented using either the Householder reflections or the Givens rotations instead.

After computing $\boldsymbol{\delta}$, the new linearization point becomes $\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i \oplus \boldsymbol{\delta}$, with \oplus being the vectorial composition operator. The nonlinear solver iterates until the norm of the correction becomes smaller than a tolerance or the maximum number of iterations is reached. This is essentially the Gauss-Newton algorithm.

The process of assembling and solving very large linear systems can become very expensive as the size of the problem grows. The employed data structure has to allow both, efficiently re-building the system every time a new linearization point is available and high speed arithmetic operations.

Cholesky factorization requires the matrix to be a symmetric positive definite one, which holds for the LS matrices.

2.1.1 *Avoiding Local Minima*

While the methods described here are attempting to avoid local minima, reaching the global minimum is not guaranteed – they are just more robust against getting trapped in a local minimum.

The process of solving nonlinear least squares is not always guaranteed to reach the global minimum and indeed the convergence depends on both the initialization of the variables [28] and on the optimization method employed. Some domains such as estimation on nonlinear manifolds contain plentiful local minima [157, 87] and more robust methods should be used, e.g. Levenberg-Marquardt [125] or Dogleg [141] algorithms. The quality of the solution is directly proportional to the sum of squared Mahalanobis norms of the residuals:

$$\chi^2 = \frac{1}{|z| - |\theta| - 1} \sum_{k=1}^m \|r_k\|_{\Sigma_k}^2, \quad (2.12)$$

where $|z|$ is the dimension of the measurements and $|\theta|$ is the dimension of the variables; this quantity indeed approaches chi-squared distribution.

The Levenberg-Marquardt [125] adds a damping factor λ to the least squares formulation, so that (2.10) becomes:

$$(\Lambda + \lambda I) \delta = \eta \quad \text{or} \quad (2.13)$$

$$(\Lambda + \lambda \text{diag}(\Lambda)) \delta = \eta, \quad (2.14)$$

where either an identity matrix I or the diagonal of Λ are used as an additive damping. Setting $\lambda = 0$ yields Gauss-Newton solution. Conversely, setting $\lambda \rightarrow \infty$ yields a solution which approaches the steepest gradient descent direction while the step size approaches zero. There are different strategies for choosing the damping factor which also depend on the choice of the damper. For equation (2.13), λ may be chosen as a product of a carefully chosen constant (e.g. 10^{-5}) and the maximum absolute value of the diagonal elements in Λ [118].

The χ^2 is calculated before and after the linearization point change and based on its increase or decrease the damping is either increased, to yield a smaller optimization step in order to avoid stepping into a local minimum or decreased in order to speed up convergence, respectively.

The Dogleg algorithm which was first described by Powell [141] uses a slightly different method to implement the same strategy. It is possible to calculate:

$$\Lambda \delta_{\text{GN}} = \eta \quad \text{and} \quad \delta_{\text{sd}} \approx \eta, \quad (2.15)$$

both the Gauss-Newton step and also the *direction* of the steepest descent step at the same time. To get the exact value of the steepest descent step, one needs to calculate the appropriate scaling factor:

$$\delta_{\text{sd}} = \alpha \eta, \quad (2.16)$$

$$\alpha = \frac{\|\eta\|^2}{\|J\eta\|^2}. \quad (2.17)$$

At this point, the trust region radius Δ is defined, which serves a similar purpose as the damping parameter λ in the Levenberg-Marquardt algorithm. This radius effectively sets the step size, and also affects the step direction, according to:

$$\delta_{dl} = \begin{cases} \delta_{GN} & \text{if } \|\delta_{GN}\| \leq \Delta \\ \delta_{sd} \frac{\Delta}{\|\delta_{sd}\|} & \text{if } \|\delta_{GN}\| \geq \Delta \\ \delta_{sd} + \beta(\delta_{GN} - \delta_{sd}) & \text{otherwise,} \end{cases} \quad (2.18)$$

where β is chosen so as to make the step size $\|\delta_{dl}\|$ equal to Δ . This gives rise to a quadratic equation where the single root of interest can be recovered using a simple analytic solution (while taking care to avoid a loss of precision):

$$c = \delta_{sd}^\top (\delta_{GN} - \delta_{sd}) \quad (2.19)$$

$$\beta = \begin{cases} \frac{-c + \sqrt{c^2 + \|\delta_{GN} - \delta_{sd}\|^2 (\Delta^2 - \|\delta_{sd}\|^2)}}{\|\delta_{GN} - \delta_{sd}\|^2} & \text{if } c \leq 0 \\ \frac{\Delta^2 - \|\delta_{sd}\|^2}{c + \sqrt{c^2 + \|\delta_{GN} - \delta_{sd}\|^2 (\Delta^2 - \|\delta_{sd}\|^2)}} & \text{otherwise,} \end{cases} \quad (2.20)$$

and thus the Dogleg step can be calculated and taken. Similarly as in Levenberg-Marquardt, the trust region radius is modified based on the improvement of the solution once the step has been taken.

The famous study of Lourakis et al. [115] shows that for vision problems, Dogleg converges faster than Levenberg-Marquardt while giving solutions of the same quality. In addition to that, Dogleg is appealing from the incremental solving point of view, as it does not require modification of the system matrix by damping which would impede incremental factorization updates. Similarly, Dogleg is favorable if not only the state mean but also state covariance is needed; then the factorization can be inverted whereas in Levenberg Marquardt, a second factorization without the damping needs to be formed first.

In incremental solving, we will attempt to only update the variables which are changing. Altering the entire diagonal of the system matrix would require recalculating much more.

2.1.2 Dealing with Outlier Measurements

In some problems, perhaps especially in computer vision, a situation often arises that some of the measurements introduced into the system are not affected by normal distributed noise, as assumed in (2.3) but rather a few of them have a significantly larger error. It is possible to introduce additional variables to the optimized system, which decide on the validity of the measurements [163]. Alternatively, it is possible to calculate the weights directly, without any additional variables as in [1] or by the use of standard *robust estimators*.

Table 2.1: A few of the commonly used robust functions. Note that a , b and c are constant parameters of the individual functions (i.e. not the same variable).

	loss function $\rho(u)$	score function $\psi(u) = \frac{\partial \rho(u)}{\partial u}$
Ordinary LS	$\frac{1}{2}u^2$	u
Huber [88]	$\begin{cases} \frac{1}{2}u^2 & \text{if } u \leq a \\ \frac{1}{2}a(2 u - a) & \text{otherwise} \end{cases}$	$\begin{cases} u & \text{if } u \leq a \\ a \operatorname{sign}(u) & \text{otherwise} \end{cases}$
Cauchy [83]	$\frac{a^2}{2} \log \left(1 + \left(\frac{u}{a} \right)^2 \right)$	$\frac{u}{1 + \left(\frac{u}{a} \right)^2}$
Tukey [15]	$\begin{cases} \frac{a^2}{6} \left(1 - \left(1 - \left(\frac{u}{a} \right)^2 \right)^3 \right) & \text{if } u \leq a \\ \frac{a^2}{6} & \text{otherwise} \end{cases}$	$\begin{cases} u \left(1 - \left(\frac{u}{a} \right)^2 \right)^2 & \text{if } u \leq a \\ 0 & \text{otherwise} \end{cases}$
Hampel [78]	$\begin{cases} \frac{1}{2}u^2 & \text{if } u < a \\ a u - \frac{1}{2}a^2 & \text{if } a \leq u < b \\ a \frac{c u - \frac{1}{2}u^2}{c-b} - \frac{7}{6}a^2 & \text{if } b \leq u < c \\ a(b+c-a) & \text{otherwise} \end{cases}$	$\begin{cases} u & \text{if } u < a \\ a \operatorname{sign}(u) & \text{if } a \leq u < b \\ a \frac{c \operatorname{sign}(u) - u}{c-b} & \text{if } b \leq u < c \\ 0 & \text{otherwise} \end{cases}$

The appealing property of robust estimators or M-estimators [88, 78, 162] (maximum likelihood type estimators) is their simple integration into the ordinary non-linear least squares framework. In fact, NLS is a special case of an M-estimator:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(\sum_{k=1}^m \rho \left(h_k(\boldsymbol{\theta}_{i_k}, \boldsymbol{\theta}_{j_k}) \ominus z_k \right) \right), \quad (2.21)$$

where the loss function $\rho(\cdot)$ happens to be the L2 norm or in case of (2.4), a squared Mahalanobis norm. To construct a more robust estimator, rather than to minimize the sum of squares which gets easily carried away by squares of outlier errors, it would be better to minimize e.g. the L1 cost. Unfortunately, L1 in particular is not differentiable and a slightly different approach needs to be taken: e.g. Huber [88] uses L2 norm for small errors to avoid problems with calculating the derivatives around zero but replaces its tails with that of appropriately scaled L1 norm so as to avoid discontinuities. Many other cost functions were proposed in the literature, some of them are listed in Table 2.1.

In order to be able to meaningfully set the parameters of any particular robust function, the relative efficiency of an estimator is defined as the ratio of variances:

$$e(T_1, T_2) = \frac{\mathbb{E} \left((T_2 - \boldsymbol{\theta})^2 \right)}{\mathbb{E} \left((T_1 - \boldsymbol{\theta})^2 \right)}, \quad (2.22)$$

where T_1 and T_2 are two estimators to be compared. Typically, an estimator T_1 would be compared to a least squares estimator T_2 as that is the most efficient one.

If the efficiency is unity for all θ , the estimator is considered efficient. To set the robust function parameters, one typically aims at 95% efficiency. E.g. for Huber's function this corresponds to setting $a = 1.345$, for Cauchy $a = 2.385$, for Hampel $(a, b, c) = (1.393, 2.787, 5.573)$ with $c = 2a + b$ and for Tukey it is $a = 4.685$.

Another problem in robust estimation is scale dependence; two problems with variables of different magnitude will behave differently in (2.21) and this problem is solved by adding a scale parameter:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \left(\sum_{k=1}^m \rho \left(\frac{h_k(\theta_{i_k}, \theta_{j_k}) \ominus z_k}{s} \right) \right). \quad (2.23)$$

Now by setting an appropriate scale s , it is possible to obtain robust treatments for problems of any scale. Otherwise, it would be possible for all the measurements to be misclassified as inliers if the scale was very small, or conversely as outliers if it was very large.

Therefore, the choice of s has large implications on the robustness of the estimate. It can be determined a-priori from the knowledge of the sensor characteristics or from the type of problem that is being solved. Alternatively, it is possible to use one of generic algorithms for estimating the scale. A popular procedure is Median Absolute Deviation (MAD) which is calculated as:

$$\text{MAD} = \underset{k=1}{\operatorname{median}}^m \left(|h_k(\theta_{i_k}, \theta_{j_k}) \ominus z_k| \right) = \underset{k=1}{\operatorname{median}}^m (|r_k|), \quad (2.24)$$

and the scale is then set as:

$$s = 1.4826 \text{ MAD}, \quad (2.25)$$

where the constant factor is intended to give unity scale for data with errors following the $\mathcal{N}(0, 1)$ distribution. An alternative to MAD was proposed by Huber [161, 88].

Finally, once the scale is known, it is possible to solve the robust estimation problem in (2.23) by collecting all weights $w_k = \frac{\Psi(u_k)}{u_k}$ where $u_k = \frac{r_k}{s}$ in a diagonal matrix $W = \operatorname{diag}(w_1, \dots, w_m)$ and writing the linearized form of the problem:

$$A^T W A \delta = A^T W \mathbf{b}. \quad (2.26)$$

Note that this is very similar to (2.11), with the exception of the introduction of the weight matrix W , in addition to measurement weights in Σ . This leads to the Iteratively Reweighted Least Squares (IRLS) algorithm where first the weights in W are calculated, then the system in (2.26) is solved, the linearization point becomes $\theta^{i+1} = \theta^i \oplus \delta$ and the process repeats until either the norm of δ approaches zero or the maximum number of iterations is exceeded.

E.g. for vision problems, the outlier threshold can be a fixed amount in pixels.

2.2 SIMULTANEOUS LOCALIZATION AND MAPPING

SLAM is a central problem in robotics and relates to navigation of a robot which at the same time builds the map that it uses to determine its location and to plan further movement [144], both the location and the map being unknown initially. There are many formulations of **SLAM**, to some degree dependent on the sensor used, the representation of the map and the underlying method for dealing with sensor noise. The most common sensor types include range finders (e.g. LIDAR, **RGBD** or time of flight cameras or sonar) and monocular, stereo or spherical cameras, **GPS**, **IMU**, as well as various combinations of those.

Kalman Filtering (**KF**) is an efficient method for dealing with noisy measurements of a linear variable. However, problems in robotics and computer vision are highly nonlinear due to projections and rotations and rather than baseline **KF**, its extensions are commonly used. Extended Kalman Filter (**EKF**) is a nonlinear version of **KF** which uses a linear approximation around the current linearization point, and has been popular in **SLAM** literature [35, 54, 43, 109].

One disadvantage of **EKF** is lower precision or even divergence, if the underlying model is highly nonlinear. For that, Unscented Kalman Filter (**UKF**) uses a sampling approach [92] in order to calculate the mean and the distribution of the estimate more accurately. Several **SLAM** approaches were formulated using **UKF** [119, 32, 84, 86], yielding a better run time and consistency than that of **EKF**-based approaches.

Information Filter (**IF**) is another variation where information matrix, the inverse of the state covariance matrix, is being propagated. The advantage is in simple integration of new observations as the information is additive, leading to more accurate estimates and higher stability. One disadvantage of **IFs** is the need to invert the information matrix often but despite that, **IFs** are relevant in **SLAM** [165, 166, 57].

Particle filtering is a popular method based on Monte Carlo sampling. It is very simple to implement and can inherently handle multiple hypotheses. The position of the robot (the estimated variable) is represented by a set of particles which are uniformly distributed initially, as the robot position is unknown. At each step, robot control commands are applied to all the particles which are then re-sampled using the posterior distribution of particle positions conditioned by map observations. The particles typically quickly converge to one or more clouds (hypotheses) where the robot could be located. FastSLAM and its variants [124, 145, 10, 100] are archetypal representatives of particle filter implementations.

The major disadvantage of filtering approaches is that they discard the information once it has been ingested by the filter and they fix the linearization point. As such, they can become inconsistent over time. A process in which both the poses and the map are retained and optimized jointly is sometimes referred to as

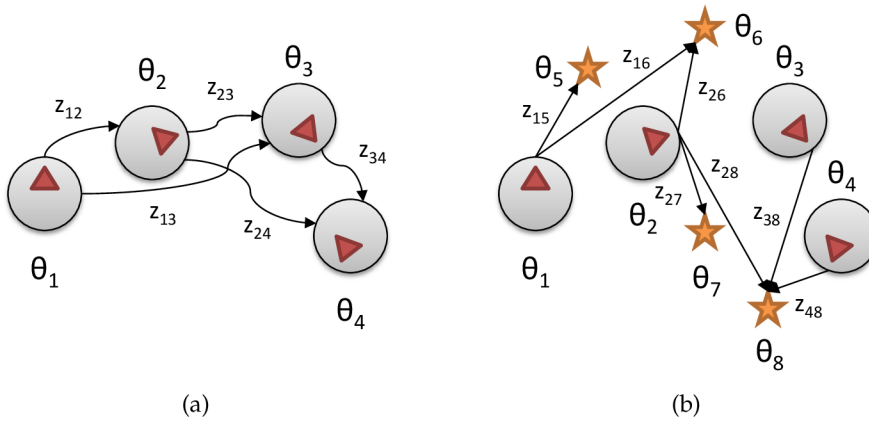


Figure 2.1: Example of a) pose and b) landmark SLAM; $\theta_1 \dots \theta_4$ are the poses of the robot, $\theta_5 \dots \theta_8$ are positions of the landmarks and $z_{i_k j_k}$ are the observations.

smoothing (as opposed to *filtering*) or Smoothing and Mapping (SAM) [45]. It has been observed that the SLAM forms a graph [9] where the optimized variables are the nodes and the observations are edges. By optimizing this graph, one obtains globally consistent maximum likelihood set of robot poses and also maximum likelihood map. Graph SLAM [167] is an unifying framework which solves the problem by variable elimination, which effectively amounts to sparse matrix factorization and standard NLS method can be used.

There are literally hundreds of extensions of this method, Pose Graph SLAM being notable in that the result is only the trajectory of the robot and the map is either represented implicitly or not recovered at all, see Figure 2.1a. Conversely, Landmark SLAM recovers the trajectory of the robot, along with positions of the landmarks in the environment, a similar example is given in Figure 2.1b. Perhaps also interesting from the point of view of this thesis are the ones that relate to explicit distributed processing, such as Tectonic SLAM [131] or similar approaches [132, 99].

In this thesis, several novel highly efficient SLAM algorithms using the Graph SLAM formulation will be described, which previously appeared in [PŠI⁺13b, PŠI⁺13a, PIŠ⁺13b, IPŠ⁺15].

2.3 BUNDLE ADJUSTMENT AND STRUCTURE FROM MOTION

Bundle Adjustment (BA) and Structure from Motion (SfM) are computer vision problems in which the 3D reconstruction of the scene is calculated. A typical sparse 3D reconstruction pipeline consists of several stages: first, visual features [116] are extracted from the images which are then matched using approximate nearest neighbor search [126] and subsequently pruned using Random Sample Consensus (RANSAC) along with geometric estimation [134]. Depending on the scale of the problem, the matching can be either done in all-to-all manner or hierarchi-

The poses can be represented e.g. as $\mathbb{R}^{4 \times 4}$ matrices and the landmarks can be e.g. \mathbb{R}^3 vectors calculated from corner points in the images or similar corner-like features in the laser scan (the famous Victoria Park [133] dataset uses tree trunks).

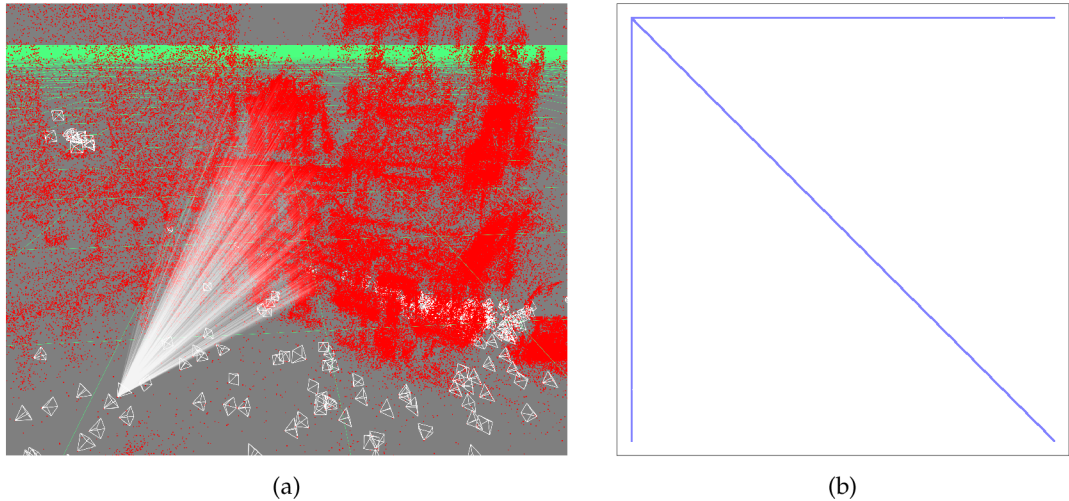


Figure 2.2: Graph of the *Venice* dataset a) edges connecting one of the cameras to the observed landmarks, b) the matrix of the entire dataset, with all the camera vertices ordered first. Note that each nonzero is inflated so as to be visible. There is deliberately space left between the border and the matrix, to be able to better see the fine arrow-like pattern.

The relative pose of two cameras can be calculated either from discrete feature points e.g. by solving the perspective three point (P3P) problem [61], or by minimizing a photogrammetric error directly on the images.

cally using approximate clustering first and then fine grained matching within the clusters [4]. The camera poses are given by relative transformations between the matched images, the landmark positions are given by triangulation of the matched feature points. Because of different sources of errors, the initial geometric estimate tends to be rather noisy and simply concatenating the calculated pose transformations and triangulating the observed feature points as they come would quickly diverge catastrophically. Therefore, one more crucial step is employed: the nonlinear optimization.

Both **BA** and **SfM** deal with noise much like **SLAM**, to which these problems are mathematically equivalent. Although the matters are perhaps more complicated, we refer as **BA** to problems dealing with unstructured databases of images – often from multiple different cameras with potentially unknown parameters and as **SfM** to problems of reconstruction from an ordered sequence of images from a single moving camera – possibly a video-sequence of smooth motion. This makes the two different from the image track processing point of view but very alike from the optimization point of view.

The distinguishing trait that sets **BA** apart from **SLAM** is the space where the error is minimized: in **SLAM**, the space in which the measurements (and thus the error) is defined is the same as that of the poses. On the other hand, in **BA**, the error of the reprojection in 2D image space is minimized while the poses and landmarks exist in 3D space.

Most of the 3D reconstruction implementations work incrementally: only one or a few frames are integrated at a time, followed by a **BA** step. Bundle adjust-

ment finds the [MLE](#) of the camera poses and the structure, given the observations and is most commonly solved using nonlinear least squares optimization. To solve the nonlinear least squares, Conjugate Gradient ([CG](#)) or a direct solver can be employed. While [CG](#) is often claimed as a linear cost algorithm [[26](#), [57](#)] it usually takes more iterations of the nonlinear solver to converge, being effectively slower than direct methods [[169](#)]. Other approaches [[169](#)] are possible.

The seminal paper [[113](#), [114](#)] describes design and implementation of an efficient [BA](#) package called SBA with the basic traits shared by most of the other implementations. The problem is formulated as a Levenberg-Marquardt [[125](#)] nonlinear least squares optimization. It makes use of the problem sparsity: not all of the points are observed by all of the cameras and the system graph is usually far from being fully connected. It also makes use of the characteristics of the [BA](#) problem which typically contains a relatively large amount of landmarks that have no relations among themselves (from the graphical point of view, they form a large independent set, or – if there are no connections among the cameras either – the entire graph is bipartite). As seen in [Figure 2.2b](#), this gives rise to diagonal sub-matrices that make the underlying linear problem easier to solve using the Schur complement [[178](#)] techniques, as opposed to applying a general linear solver directly to the whole matrix.

Photo tourism [[155](#)] is an application of [BA](#) to building sparse reconstructions from unstructured collections of photographs for the purpose of interactive navigation in such collection. It makes use of the EXIF image tags to get the intrinsic camera parameters rather than solving an uncalibrated problem. The camera poses are primarily used for photo placement in the user interface while the sparse structure is rendered as textured points (rather than performing triangulation), optionally in non-photo-realistic mode. Two techniques for view interpolation for animated transitions are suggested.

Using unstructured photograph collections from the Internet allows for extremely large scale 3D reconstruction [[4](#)]. The problems that need to be tackled exist both in the vision part of the reconstruction pipeline as well as in the [BA](#) optimizer. In the vision part, the feature matching becomes the bottleneck, as it scales with $O(n^2)$ in the number of images and hierarchical matching is proposed to solve the problem both more efficiently and in parallel. Agarwal et al. [[4](#)] implement two optimization strategies which are selected based on problem size. The first one is a block diagonal preconditioned Conjugate Gradient ([CG](#)) solver. The second one is rather similar to SBA, with the difference that unlike in SBA [[113](#), [114](#)] where the Schur complement is solved using dense LDL^T factorization, a sparse Cholesky factorization is employed here to gain up to an order of magnitude speedup for large systems where the Schur complement is quite sparse. Similar speedups were reproduced by e.g. [[105](#)].

This also depends on the preconditioning of the system, an advanced topic which is not discussed here as this thesis relates more to direct solvers.

The authors actually do not specify which one is used for small problems.

To further accelerate the optimization part, it is possible to employ parallelization. **CG** solvers are parallelized easily, as they basically only require parallel implementation of sparse matrix-vector multiplication routine [176]. For direct solvers, distributed optimization techniques were proposed [130] where the problem is split into several sub-problems with minimal graph separators that are solved independently, followed by a separator optimization pass. Such methods can be easily used for parallelization on clusters.

Parameterizations taking advantage of the incremental solving were proposed as well. In [153], relative camera and pose formulation is employed, rather than using a single global Euclidean coordinate frame. After adding a new camera pose and the associated observations, it is possible to find the variables where this addition induced a significant change and only a reduced system consisting of those variables and their neighbors is solved. The size of the system that needs to be solved is only a fraction of the full system, making the optimization faster. A standard Schur complement solver is employed.

Another approach is acceleration via graph sparsification. In [91], rather than optimizing the entire problem only the camera poses are optimized, with the observations taking form of three-view constraints related to the tri-focal tensor. A similar generalized approach is proposed in [27] where the structure variables would be represented implicitly by the corresponding triangulation functions and therefore only the camera poses and optionally also their calibrations would be optimized. In both cases, the structure points can be triangulated after-the-fact in the least squares fashion from all the cameras that observe each given point. Since these methods effectively solve a pose graph, it is possible to use the appropriate incremental algorithms [95, 98], [PIŠ⁺13b] as well.

2.4 FINITE ELEMENT METHODS

FEM is a class of popular methods used in physics simulation. While less related to **SLAM** or **BA** by the underlying principles of estimation, they also feature graphical structure. This yields matrices with certain sparsity patterns with characteristics not unlike those of **SLAM** or **BA**, and the basic matrix operations described in this thesis are also useful in solving **FEM** problems.

Since the domain of the real world is continuous, it can be difficult to parametrize and describe it numerically for the needs of physics simulation. **FEM** sets out to solve this issue by discretizing the simulation domain into a large (but finite) number of elements. Typically, those can be triangles, quadrilaterals or tetrahedra, which are connected in a mesh. Formally, **FEM** solves a linear system:

$$Ku = f, \quad (2.27)$$

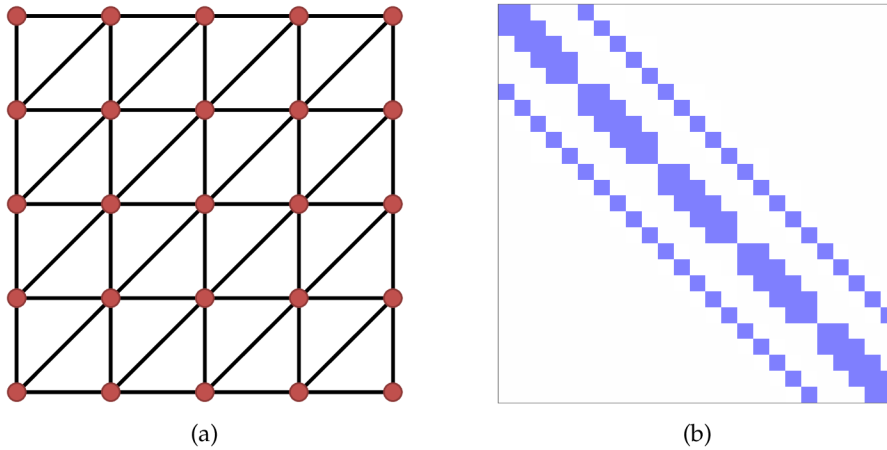


Figure 2.3: Example of a) 5×5 uniform FEM mesh and b) the associated stiffness matrix.

where K is the stiffness matrix, f is the load vector and u holds the unknown element states which we seek. The stiffness matrix is given by a sum of the stiffness matrices of the elements, which are themselves derivatives of basis functions which describe the elements, integrated over each element area. The choice of the basis function is a trade-off between accuracy and computational complexity and there are many functions used in practice [67, 158], some of which yield dense blocks. An example of a stiffness matrix for a small triangular mesh is given in Figure 2.3b. Note that each of the nonzeros could be a dense block, depending on the choice of the basis functions. For example, assuming 2D triangular mesh and piecewise linear basis functions, the element stiffness matrix is a 2×2 dense matrix.

Furthermore, [158] describes a special case of the stiffness matrix which is applicable to structured 2D grids. It relies on writing the system in the following block tridiagonal form:

$$\begin{pmatrix} A & -T & & & \\ -T & A & -T & & \\ & \ddots & \ddots & & \\ & & & -T & A \end{pmatrix} \cdot \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_N \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_N \end{pmatrix}. \quad (2.28)$$

Assuming a few conditions are met ($AT = TA$ and $N = 2^{s+1} - 1$ for some integer s), it is possible to solve such a system using the cyclic reduction method [158] which requires s modifications of the system to arrive at a single block equation which can be solved using e.g. Gaussian elimination and then one can work back to recover the rest of the solution. It would be equally possible to employ sparse block matrices to calculate the solution using a (more) direct method.

SPARSE MATRIX REPRESENTATIONS

The problems described in the previous sections (among others) all have a graph structure which can be represented as a sparse matrix and matrix operations are used to find solutions to those problems. In the field of numerical techniques and matrix computations, Basic Linear Algebra Subprograms (BLAS) is the de-facto standard for the implementation of matrix representations, as well as the interface for the matrix operations. While the original BLAS proposal [108, 46, 47] was for dense matrices, it was later adopted for sparse matrices as well [49]. The sparse BLAS proposal specifies several sparse matrix storage formats, some of which will be briefly revised below. Unless specified otherwise, the formats are elementwise.

3.1 COORDINATE FORMAT

The coordinate format, often abbreviated C00, is a very simple sparse matrix format; it stores each nonzero entry as a triplet of *row*, *column* and the associated *value*, with no ordering imposed by a rule or at least a convention. It is suitable for assembly of the sparse matrices and it is simple to erase or add more values at any time. For the matrix in Figure 3.1a, the C00 representation is in Listing 3.1.

To better illustrate the properties of this format, let us consider a simple matrix vector product of the form $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{y}$, listed in Algorithm 3.1. Looking at this procedure, two things quickly become apparent. In case there are several entries for the same row and column, they will have the same effect as if those duplicate entries were summed up into a single one. This is a convention commonly observed

$$\begin{array}{c}
 \left[\begin{array}{cc} 6 & 4 \\ 7 & \\ & 9 & 4 \\ 2 & 5 & 3 \\ 2 & & 1 \\ & 1 & 2 \end{array} \right] \\
 \text{(a)}
 \end{array}
 \quad
 \begin{array}{c}
 \left[\begin{array}{ccc} 6 & 5 & \\ & 1 & 7 \\ & & 9 & 4 \\ & & & 3 \end{array} \right] \\
 \text{(b)}
 \end{array}$$

Figure 3.1: Example sparse matrices.

Algorithm 3.1: A basic matrix - vector multiplication algorithm.

```

1: function COO_GEMV(A, x, y)
    ▷ calculates  $\mathbf{y} = A\mathbf{x} + \mathbf{y}$  where  $\mathbf{x}$ ,  $\mathbf{y}$  are dense and  $A$  is sparse in COO format
2:   for i = 0 to A.nnz - 1 do
3:     r = A.row[i]
4:     c = A.col[i]
5:     y[r] = y[r] + A.val[i] · x[c]
6:   end for
7: end function

```

for the COO format. A second issue is regarding the performance of the algorithm: the elements of the \mathbf{x} and \mathbf{y} vectors are both indexed by the sparsity pattern of A , the algorithm therefore both *gathers* and *scatters* elements in memory which can lead to poor performance if the entries are ordered unfavorably. Since the matrix vector product is a building stone of the CG solvers, more efficient formats were proposed.

Listing 3.1: The matrix from Figure 3.1a stored in the COO format.

```

1: m = 6; n = 4; nnz = 12;
2: row = {0, 0, 1, 2, 2, 3, 3, 3, 4, 4, 5, 5};
3: col = {0, 3, 0, 2, 3, 0, 1, 3, 0, 3, 1, 3};
4: val = {6, 4, 7, 9, 4, 2, 5, 3, 2, 1, 1, 2};

```

3.2 SPARSE DIAGONAL

The diagonal format (DIA, not to be confused with a diagonal *matrix*) strives to represent the sparse matrix in a more structured manner. It consists of the array of several dense diagonals and an array of their offsets, as illustrated in Listing 3.2. The matrix vector product now degenerates to a sum of dense dot products which are implemented efficiently on today's machines.

Note that the matrix in Figure 3.1b has three nonzero diagonals and thus the diag array in Listing 3.2 has only three rows, not four.

Listing 3.2: The matrix from Figure 3.1b stored in the DIA format.

```

1: m = 4; n = 4; ndiag = 3;
2: diag = {6, 1, 9, 3, // The * entries are outside of the matrix and their
3:         0, 7, 4, *, // value does not matter (but they are stored).
4:         5, 0, *, *}; // Also note the filled-in zeros.
5: ioff = {0, 1, 2}; // Diagonals below the main would have negative offsets.

```

Matrices consisting of just a few diagonals arise in some applications and then the diagonal format is suitable. Matrices with just a single diagonal are often repre-

sented just as a dense vector with special semantics. Consider, however, storing the matrix in [Figure 3.1a](#): all the diagonals but one are occupied and the matrix would in effect be stored almost as a dense matrix, with many zero entries represented explicitly. For that reason, the diagonal format is often paired with a different format so that the almost empty diagonals can be represented more efficiently. This is often referred to as a hybrid (HYB) format.

3.3 SKYLINE

Triangular matrices can be represented using the skyline (SKY) format, symmetric matrices can be similarly stored as a triangular matrix with symmetry semantics, sometimes also referred to as the symmetric skyline (SSK) format. Assuming an upper-triangular matrix, skyline stores rows of the matrix, starting with the diagonal element and ending with the last non-zero element in the given row. For lower-triangular matrix, the range of elements would start with the first non-zero and end with the diagonal instead.

An appealing quality of the skyline format is that operations such as Gaussian elimination or Cholesky factorization do not change the structure of the matrix and can be performed in-place. The obvious disadvantage is that it is only efficient for matrices with all the non-zero elements situated close to the diagonal. An example of an upper-triangular skyline matrix is shown in [Listing 3.3](#).

Listing 3.3: The matrix from [Figure 3.1b](#) stored in the SKY format.

```

1: m = 4; n = 4;
2: val = {6, 0, 5, // Note the filled-in zero.
3:      1, 7,
4:      9, 4,
5:      3};
6: rptr = {0, 3, 5, 7, 8}; // Array of row beginning / end pointers.
```

3.4 ELLPACK-ITPACK

The Ellpack format (ELL) is conceptually similar to the diagonal format but is geared towards general sparse matrices without a prominent diagonal structure. It relies on the number of nonzero entries per row being relatively similar over the whole matrix. The matrix is represented by a pair of a dense matrix containing the values of each row with the zeros removed and a corresponding column permutation matrix of the same size. The number of rows in these matrices matches the

original matrix and the number of columns is given by the maximum number of entries in a single row. An example of this is shown in [Listing 3.4](#).

Listing 3.4: The matrix from [Figure 3.1a](#) stored in the ELL format.

```

1: m = 6; n = 4; ndiag = 3;
2: coef = {6, 7, 9, 2, 2, 1,
3:       4, 0, 4, 5, 1, 2,
4:       0, 0, 0, 3, 0, 0}; // Note the filled-in zeros in shorter rows.
5: col = {0, 0, 2, 0, 0, 1, // The * entries do not correspond to any
6:       3, *, 3, 1, 3, 3, // non-zero value and can point to arbitrary
7:       *, *, *, 3, *, *}; // column (e.g. the last one - 3).

```

The matrix-vector product is again implemented as a sum of dot products, with the modification that for each dot product the right-hand side vector needs to be gathered from memory based on the column indices. This was implemented as an instruction in vector processors so that it could be implemented efficiently [146]. This operation can also be implemented on a GPU, with performance depending on the characteristics of the data.

3.5 JAGGED DIAGONAL

A basic disadvantage of the Ellpack format is the reliance on uniform row lengths. For the matrix in [Figure 3.1a](#), 33% of zeros are filled in because the third row is longer than the others. The Jagged Diagonal (JAD) further improves upon Ellpack, for matrices with uneven distribution of non-zeros and also keeps parallel processing on vector processors in mind. First, the rows of the matrix are sorted by descending number of non-zeros ([Figure 3.2b](#)) and the permutation for obtaining the original matrix is recorded ([Figure 3.2c](#)). Then the rows are compacted similarly

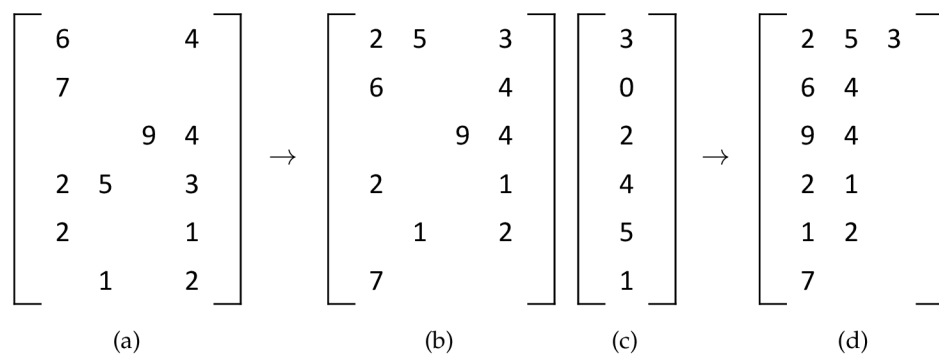


Figure 3.2: Converting a matrix to the jagged diagonal format: a) the original matrix, same as in [Figure 3.1a](#), b) sorted by row lengths, c) the associated permutation vector and d) compacted rows are stored column-wise as three "diagonals").

as in the Ellpack format (Figure 3.2d) and the column indices are also recorded. To avoid storing zero entries, the numbers of nonzero elements for each row are recorded as well. The matrix stored in the JAD format is shown in Listing 3.5.

Listing 3.5: The matrix from Figure 3.1a stored in the JAD format.

```

1: m = 6; n = 4; ndiag = 3; nnz = 12;
2: val = {2, 6, 9, 2, 1, 7,
3:       5, 4, 4, 1, 2,
4:       3};
5: col = {0, 0, 2, 0, 1, 0,
6:       1, 3, 3, 3, 3,
7:       3};
8: ilg = {3, 2, 2, 2, 2, 1}; // row lengths
9: dlgl = {6, 5, 1}; // diagonal lengths
10: perm = {3, 0, 2, 4, 5, 1}; // row permutation to yield the original matrix

```

The matrix vector multiplication for the JAD format would be implemented similarly as in the case of Ellpack, via dense vector dot products with right hand side gathering. In addition, the left hand side vector needs to be permuted at the beginning and inversely permuted at the end – but that is a fast linear time operation. On the other hand, there are no filled-in zeros and no computation is wasted. The format was designed for highly efficient operations on vector processors.

3.6 COMPRESSED SPARSE COLUMN OR ROW

Compressed sparse column (CSC) and its transpose, compressed sparse row (CSR), are formats aimed at storing general sparse matrices while being able to perform operations efficiently. CSC stores the non-zeros of the matrix column by column and ordered by row in terms of each individual column (although notable exceptions exist [41]). Along with the non-zeros, an array of row indices is stored. So far, the format is equivalent to the C00 format. But rather than storing column indices for each element, column pointers are stored instead (either in a single array of pointers to the first element with the total number of elements appended at the end, or in a pair of index arrays of the first elements of each column and of the last elements of each column).

That makes it easy to access the columns of the matrix in any order while the rows must be accessed sequentially. The algorithms that work with these formats order their loops so that row lookup is avoided, for greater efficiency. The format is geared more towards scalar processors; the matrix - vector product cannot be formulated in terms of dense dot products as it requires both gather and scatter operations. In contrast to the C00 format, the memory accesses are predictable

Compared to the triplet format, the compression ratios achieved on common matrices average at about 1.45 : 1, peaking at slightly less than 2 : 1, which is the upper bound.

and fast algorithms exist [41]. An example of CSC representation is in Listing 3.6. Modified formats based on CSC and either a dense diagonal vector or several dense diagonals in the DIA format exist.

Listing 3.6: The matrix from Figure 3.1a stored in the CSC format.

```

1: m = 6; n = 4;
2: val = {6, 7, 2, 2, 5, 1, 9, 4, 4, 3, 1, 2};
3: row = {0, 1, 3, 4, 3, 5, 2, 0, 2, 3, 4, 5};
4: cptr = {0,          4,    6, 7,          12}; // column pointers, nnz

```

3.7 BLOCK COMPRESSED SPARSE ROW

Matrices which have *sparse block* structure appear in many applications, notably in those described in Sections 2.2, 2.3 and 2.4. These matrices can be readily stored using the general-purpose *elementwise* (sometimes also *point*) sparse formats but there are some advantages in exploiting the block structure explicitly. Block compressed sparse row (BSR) and its less common transpose, block compressed sparse column (BSC), are extensions of the corresponding elementwise sparse formats CSR and CSC, respectively. Multiple other variants of these block formats exist, e.g. block coordinate (BCO), block sparse diagonal (BDI) or block Ellpack (BEL). For symmetric matrices, the convention is that the diagonal blocks store both the upper and the lower half.

Mean compression ratios compared to the triplet format are 2.69 : 1, with upper bound 3 : 1 (regardless of block size). This is on matrices with no 1×1 blocks.

Elements become *blocks*, columns and rows become *block columns* and *block rows*, respectively. These formats assume that all the blocks in the matrix are square and have the same size. Each block is stored in either row-major or column-major (depending on the implementation convention), including any zeros. Two examples of block matrices are given in Figure 3.3, while 3.3a (two block columns and two block rows, three nonzero blocks) will need to store some zeros. The other matrix

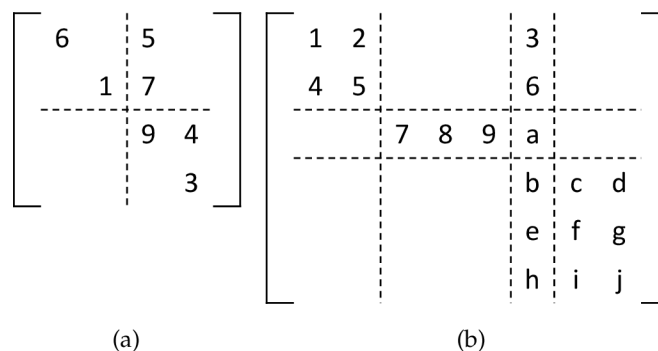


Figure 3.3: Example sparse block matrices.

in [Figure 3.3b](#) (four block columns, three block rows, six nonzero blocks) does not have any zeros in its conceptual blocks, but they are not of the same size which is not suitable for the BSR format – the block boundaries would have to be moved in order for all the blocks to be the same size (e.g. 2×2) and some additional zeros would be stored again. An example of a matrix stored in the BSR format is in [Listing 3.7](#).

It is notable that compared to the representation of the same matrix using the CSR format, the index arrays take up 50% less space and similarly the algorithms will spend less time in the indexing and control logic parts and more in the computational parts (CSR would need 5 row pointers and 7 column indices). Not all matrices can benefit from block schemes though, e.g. the matrix from [Figure 3.1a](#) would end up being completely dense if represented using 2×2 blocks.

Listing 3.7: The matrix from [Figure 3.3a](#) stored in the BSR format.

```

1: m = 4; n = 4; // four rows and four columns
2: bm = 2; bn = 2; // two block rows and two block columns
3: nnz = 12; bnnz = 3; // twelve non-zeros, three non-zero blocks
4: lb = 2; // the blocks are 2 x 2
5: val = {6, 0, 0, 1, 5, 7, 0, 0, 9, 0, 4, 3}; // block data (column-major)
6: bindx = {0,          0,          1}; // per-block block column indices
7: bpntrb = {0,          2}; // the first block in each block row
8: bpntrc = {          2, 3}; // one past the last block

```

3.8 VARIABLE BLOCK COMPRESSED SPARSE ROW

While the previous section introduced formats which assume all the blocks to be the same size, many applications will require matrices with rectangular blocks or mixtures of different block sizes. One approach, referred to as *splitting* in the literature, which is indeed sometimes used in solving FEM [168, 139, 65] is to represent such matrix as a sum of several matrices, each containing blocks of one particular size. That works well for operations with additive semantics, such as matrix vector products, but it is not a viable alternative for e.g. Gaussian elimination or matrix factorizations. Therefore, a variable block compressed sparse row (VBR) format was introduced. It is an extension of the BSR format with additional information about the sizes of the blocks and the layout of the block rows and block columns. An example of such a matrix is in [Listing 3.8](#).

It is notable that there are few implementations of the VBR scheme, perhaps due to its complexity and algorithmic overheads (intuitively, there are more nested loops required for the implementation of arithmetic operations which cannot be

Mean compression ratio is slightly lower than that of BSR, about 2.35 : 1. The upper bound is the same (3 : 1).

unrolled). This results in diminished performance unless the blocks are very large. To our best knowledge, the only such implementation is NIST Sparse BLAS¹ although few other variants of the format exist and their implementations will be discussed in the following chapter. There are more libraries that support the BSR format, most notably Intel MKL [59] or PETsc [12].

Listing 3.8: The matrix from Figure 3.3b stored in the VBR format.

```

1: m = 6; n = 10; // six rows and ten columns
2: bm = 3; bn = 4; // three block rows and four block columns
3: nnz = 19; bnnz = 6; // nineteen non-zeros, six non-zero blocks
4: rpnttr = {0, 2, 3, 6}; // rows of the block row origins
5: cpnttr = {0, 2, 5, 6, 8}; // columns of the block column origins
6: val    = {1, 4, 2, 5, 3, 6, 7, 8, 9, a, b, e, h, c, f, i, d, g, j};
7:
8:         // block data (column-major)
9: indx    = {0,          4,    6,          9,10,    13,          19};
10:        // indices of the first element of each block, nnz
11: bindx   = {0,          2,    1,          2, 2,    3};
12:        // per-block block column indices
13: bpntrb  = {0, 2, 4}; // pointers to the first block in each block row
14: bpntrt  = {2, 4, 6}; // and to one past the last block in each block row

```

¹ <http://math.nist.gov/spblas/original.html>

A BRIEF REVIEW OF EXISTING NLS SOLVERS

This chapter describes existing implementations of general Nonlinear Least Squares (NLS) solvers which are used in robotics, with the focus on the algorithms and data structures. The list is definitely not an exhaustive one, but an attempt was made to have the most significant implementations present. There is a considerable overlap with other scientific communities that solve computer vision, surveying, photogrammetry or other similar problems, often using their own methods and software tools.

4.1 TORO

TORO [71] was inspired by the influential work of Olson et al. [136] on stochastic gradient descent for map optimization in SLAM. Olson proposes an incremental pose parameterization where the state of the optimized system is given by the algebraic differences of the consecutive poses (rather than by their inverse composition). This yields simple sparse Jacobian but requires the consecutive poses to have only small rotational differences, otherwise the poses are captured imprecisely. Another difficulty with the relative parameterizations when coupled with stochastic gradient descent is that each constraint affects multiple variables, slowing down the convergence since different constraints can affect the same variables in antagonistic fashion, effectively undoing each other's work.

TORO uses a similar parameterization, but rather than using relative representation between the consecutive poses, it constructs a spanning tree and represents the poses as the algebraic difference of a pose and its parent. Therefore, a constraint between two poses goes through the common ancestor and less poses need to be updated, speeding up the convergence of the algorithm. Note that the spanning tree does not replace the system graph - it merely affects the numerical representation of the optimized variables. The proposed algorithm also allows for node reduction in case the robot navigates a previously visited area.

Written in C++, it only uses small fixed-size dense matrices since the structure of the problem and the approximations applied do not necessitate the use of sparse matrices. It can solve for 2D and with a later extension also 3D pose graphs. It is self-contained in the sense that it does not depend on any libraries e.g. for linear algebra or graph orderings.

In case of several poses, each represented relative to the previous one, then to get the absolute coordinates of the last pose, all the poses are needed. This is a common downside of relative parameterizations.

4.2 ISAM

In their work, Dellaert and Kaess [45] investigate the connection between the factor graph formulation of the SLAM problem, its matrix representation and the correspondence of the factorization of a such matrix to the variable elimination in the corresponding graph. Initial matrix factorization performance results are reported, with the goal to develop an efficient incremental NLS solver. The authors recognize the importance of the need of a good variable ordering and argue for the use of the QR factorization via Givens rotations because of the simplicity of its incremental implementation compared to Cholesky factorization up/down-dates. Interestingly, they report their implementation of sparse matrix product to be slower than the subsequent factorization of its result.

Incremental Smoothing and Mapping (iSAM) [95] is an implementation which focuses on incremental solving, using incremental QR factorization. Unlike TORO, it uses exact derivatives (or automatically generated numerical ones) and so the Jacobian does no longer consist of identity matrices and sparse matrix computations are employed throughout. 3D rotations are expressed and optimized either in the domain of Euler angles (yaw, pitch, roll) or as quaternions.

The solver maintains QR factorization incrementally, adding only entries for the new observations while the changes in the rest of the Jacobian are ignored. Every one hundred steps, the variables are reordered using the COLAMD algorithm to avoid fill-in, the Jacobian is recomputed and the whole QR factorization is calculated from scratch. This approach, although no longer exact, proves quite efficient. iSAM also provides calculation of covariances using the recursive formula [18, 68] in conjunction with dynamic programming [93] or alternatively using a fast conservative approximation described in [57].

Incremental QR factorization is a well researched subject, since it allows for out-of-core techniques which allowed tackling even large scale problems on computers with small core memory.

Initially implemented in OCaml, the currently available version is in C++ and makes use of its own custom sparse matrix representation where each column of a sparse matrix is represented using a sparse vector. This essentially corresponds to the CSC format, except that the data is not stored in a single contiguous array but is split into one array per column. This makes adding new values easier as less data needs to be shifted around.

Despite using elementwise matrices, iSAM makes a limited use of the block structure by calculating the fill-reducing ordering on variables (which corresponds to block columns) and then expanding this ordering to the individual elements (columns). It uses either CSparse [41] or Cholmod [42] internally for batch matrix factorizations – the disadvantage is that it needs to convert from its own sparse format to standard CSC before such libraries can be used, and then to convert the result back.

4.3 HOG-MAN

HOG-Man [72] is hierarchical SLAM optimizer, working on manifolds. The manifold representation is important, as representing the rotations using Euler angles (e.g. such as iSAM) introduces singularities and gimbal locks and similarly using over-parameterized representations such as quaternions easily leads to denormalization. Instead, the authors define a manifold projection operator which converts the rotation from its representation to the manifold where it is locally Euclidean and can be optimized using classical methods, such as Gauss-Newton. HOG-Man thus uses axis-angle representation to represent the rotations.

Another important feature is hierarchical graph optimization. A graph is represented at the highest level as well as on several lower levels of detail. The first level of detail is obtained by dividing vertices of the original graph into subgraphs and treating each subgraph as a vertex. For that, a representative vertex is selected from each subgraph. There is an edge between representative vertices of two graphs in case there were edges between any two vertices between those graphs. The measurement and its associated covariance are calculated by solving for a problem consisting of the union of the two graphs, with the representative vertex from the first graph being in the origin and the representative vertex from the second graph providing the mean as well as covariance. The hierarchical optimization starts at the highest level (the smallest graph) and the significant changes are propagated downwards via rigid body transformations. When needed, a lower level subgraph can be refined using another optimization round, with the additional constraint on the representative vertices. Covariances can be calculated for the data association.

Written in C++, HOG-Man uses the graph as its primary representation rather than a matrix. It stores an associative array of vertices and an ordered set of pointers to the edges. Such structure allows for simple graph modifications but requires multiple indirections to access any element of the graph. It makes use of CSparse [41] for Cholesky decomposition and solving, while the graph or its subgraphs are converted into sparse matrix form on the fly, rather than maintaining a matrix and updating it incrementally. No advantage is taken of the block structure of the matrices.

4.4 SSBA

Sparse Sparse Bundle Adjustment [105] is a bundle adjustment optimizer, similar to the one described by Lourakis and Argyros [113, 114]. It uses the Levenberg Marquardt method described in Section 2.1.1, in conjunction with the standard Schur complement trick, followed by *sparse* Cholesky factorization (rather than

dense as in case of Lourakis), using Cholmod [42]. The feature that sets it apart is that it uses its own hybrid sparse block matrix scheme where each block column consists of an associative array that contains dense blocks, indexed by rows. The diagonal blocks are stored in a separate linear array (structurally rank deficient matrices are not permitted). The format only permits blocks of one size, $\mathbb{R}^{6 \times 6}$.

However, this matrix is only maintained for convenience of formation of the Schur complement. No matrix product operation is implemented on this block matrix – the partial products are accumulated based on camera tracks. Sparse block matrix-dense vector product is implemented for the CG solver. For the direct solver, the Schur complement needs to be converted to elementwise sparse CSC format each time before solving. This conversion is split to two phases, the structure allocation and value filling, in an attempt to save some time in nonlinear iterations where the structure does not change.

4.5 SPA

Similar to sSBA, Sparse Pose Adjustment [104] reuses a slightly modified implementation of Levenberg Marquardt optimizer to solve 2D pose-SLAM problems. It forms the information matrix (rather than the Schur complement as in sSBA) using the same sparse block matrix structure. Another interesting feature is that the whole solver is specialized for 2D pose graphs, which means that all the block matrices are $\mathbb{R}^{3 \times 3}$ and dynamic memory allocation can thus be saved, resulting in better performance. Still, this solver also takes advantage of elementwise sparse Cholesky factorization implemented in CSparse [41] and no computation is actually saved by using sparse block matrices.

4.6 G2O

A General Framework for Graph Optimization (g2o) [106] is the culmination of the research done on sSBA and SPA and has quickly become a popular framework for nonlinear optimization in robotics. It contains several optimizers, based on Gauss-Newton, Levenberg-Marquardt or Dogleg methods. While designed to be easily extensible, g2o can solve BA and SLAM problems out-of-the-box. It uses Lie algebra [159] group $SE(3)$ to correctly calculate derivatives involving spatial rotations and optimizes such variables in the tangent space vectorial form $\mathfrak{se}(3)$. In addition, it contains numerical differentiation functions to calculate derivatives automatically if needed. It can also recover covariances of the estimate, using the same recursive formula implementation as described in [93]. The support for robust solving is also implemented.

Compared to sSBA, the sparse block matrix scheme is changed, the diagonal element storage was removed and now all the matrices are represented as a vector of block columns where each block column is row-indexed associative array of matrix blocks. This is similar to sSBA, with the exception that in g2o the blocks can take any size, including matrices with blocks of mixed size. The (incomplete, to save space) C++ prototype of the matrix storage can be seen in Listing 4.1. Several operations are implemented on this matrix format, including addition, matrix and vector products, transpose and scalar multiplication. Diagonal matrix *view* is implemented for faster access to the block diagonal (but it is not useful for representing block diagonal matrices by itself). Linear solving is accomplished using one of CSparse [41], Cholmod [42], Eigen [73] or CG. Conversion to elementwise sparse matrix is again required (except for CG) every time linear solving takes place.

Listing 4.1: The g2o sparse block matrix format (C++; the code comments were redacted).

```

1: template <class MatrixType = Eigen::MatrixXd>
2: class SparseBlockMatrix {
3: public:
4:     typedef MatrixType SparseMatrixBlock; // a single dense block
5:     typedef std::map<int, SparseMatrixBlock*> IntBlockMap; // block column
6:
7:     // [constructors and operations on block matrices]
8:
9: protected:
10:    std::vector<int> _rowBlockIndices; // cumulative sum of block rows
11:    std::vector<int> _colBlockIndices; // cumulative sum of block columns
12:    std::vector<IntBlockMap> _blockCols; // block columns as assoc. arrays
13: };

```

Notably, g2o allows for optimization of problems with fixed-size blocks, with the limitation that only a single block size is supported, or the block size can be different for poses and for landmarks but then the landmarks are either handled by the Schur complement or are not optimized at all. This allows for highly efficient solving because then the inner loops in all the block matrix operations can be unrolled and Streaming SIMD Extensions (SSE) vectorization can be applied. Solvers for 2D SLAM ($\mathbb{R}^{3 \times 3}, \mathbb{R}^{2 \times 2}$), 3D SLAM or BA ($\mathbb{R}^{6 \times 6}, \mathbb{R}^{3 \times 3}$) and Sim(3) BA ($\mathbb{R}^{7 \times 7}, \mathbb{R}^{3 \times 3}$) are instantiated by default.

The iSAM2 algorithm [98] implemented in the GTSAM library, is an improvement over the iSAM solver. It uses a novel data structure called the Bayes Tree, which is

a graphical representation of the square root matrix (the Cholesky factorization of the information matrix). It allows for incremental variable reordering and selective relinearization which was not previously implemented on matrices.

The solving process in iSAM2 is a three stage one, starting by eliminating the factors from a factor graph to yield a Bayes net, then turning this Bayes net into a Bayes tree and finally solving by backsubstitution. The step of turning a Bayes net (a chordal graph) into a Bayes tree (a directed clique graph) is done in reverse elimination order and thus the information in the tree is propagated towards the root. When some variables are changed, the root of the tree and the descendants on the path to the affected variables need to be recalculated, the unaffected children are then re-attached.

The solving is thresholded by a small constant on backsubstitution (which can skip cliques of the Bayes Tree where the change in the solution is to be low). Thanks to that, the backsubstitution usually runs in better than linear time. Another several orders of magnitude larger, threshold is on the increment of the variables to be relinearized. The relinearization is only performed every ten steps of the algorithm by default. Constrained column Approximate Minimum Degree (AMD) (CCOLAMD) is employed for variable ordering, with the most recent variables ordered last in order to reduce the size of the incremental updates (since the new observations are most likely to reference those variables). Unlike iSAM, more precise Lie-algebraic derivatives calculated using the exponential map paradigm are employed throughout.

Written in C++ with heavy use of BOOST¹, it has a limited implementation of *dense* block matrices with no arithmetics support, except for operations on the individual blocks and dense Cholesky factorization. It uses Eigen [73] for linear algebra. The algorithm to calculate the marginal covariances of the variables has changed since iSAM, to one using the Bayes Tree instead of the recursive formula. It is only efficient in case a covariance of a single variable is sought after – there is no efficient way of recovering covariances of multiple variables at once [IPŠ⁺15].

4.8 CERES

Google’s Ceres solver [3] received much attention, as it is used in their 3D Maps and Street View applications. It is mostly focused on batch solving, using a variety of available algorithms (Gauss-Newton, Levenberg-Marquardt, Powell’s Dogleg, subspace Dogleg [25], CG, BFGS and LBFGS). It relies on SuiteSparse [41] and Eigen [73] for solving the linear systems via a set of sparse and dense solvers. It supports automatic and numeric derivatives, as well as analytical ones. It also

¹ <http://www.boost.org/>

has a multitude of robust loss functions. Ceres can also recover covariances of the solution, either using dense SVD or using sparse QR decomposition followed by sparse right-hands-side backsubstitution.

Listing 4.2: The Ceres sparse block matrix format (C++; the constructors and member functions omitted, the code comments were redacted).

```

1: struct Block {
2:     int32_t size; // number of element rows [columns]
3:     int position; // position along the row [column]
4: };
5:
6: struct Cell {
7:     int block_id; // block column [block row] id
8:     int position; // offset to the values_ array (see below)
9: };
10:
11: typedef struct CompressedList {
12:     Block block; // offset and height [width], in elements
13:     std::vector<Cell> cells; // a list of dense blocks
14: } CompressedRow, CompressedColumn;
15:
16: struct CompressedRowBlockStructure {
17:     std::vector<Block> cols; // description of block columns
18:     std::vector<CompressedRow> rows; // linear array of block rows
19: };
20:
21: class BlockSparseMatrix : public SparseMatrix {
22: private:
23:     int num_rows_, num_cols_; // size of the matrix, in elements
24:     int max_num_nonzeros_, num_nonzeros_; // capacity and size of values_
25:     scoped_array<double> values_; // values of the elements of the matrix
26:     scoped_ptr<CompressedRowBlockStructure> block_structure_; // \ldots
27: };

```

Notably, Ceres also has its own block matrix storage format. It is a bit more similar to the classical sparse matrix formats (e.g. to VBR) in the sense that it has an array for element values rather than storing block data in separate structures, e.g. as in g2o. The layout itself is based on the CSR format, with each block row consisting of a starting row, number of rows in the block and the list of blocks in that row, each block being a pair of block column index and offset to the dense block data. This structure can be seen in more detail in [Listing 4.2](#). Basic operations such as matrix-vector products, scalar products and conversion to triplet form are

Table 4.1: Overview of the state-of the art [NLS](#) solver data structures. Note that the linear solvers marked by dagger[†] require data conversion from the [NLS](#) solver internal storage format.

NLS Solver	Storage format	Linear solver
TORO	custom	stochastic gradient descent
iSAM	modified CSC	sparse QR [†]
HOG-Man	graphical structure	sparse Cholesky [†]
sSBA	hybrid sparse block	sparse Cholesky [†]
SPA	hybrid sparse block	sparse Cholesky [†]
g20	custom sparse block	sparse Cholesky [†] or CG
iSAM2	Bayes tree	variable elimination
Ceres	custom sparse block	sparse Cholesky [†] or CG

available. Although Ceres contains interface for general block matrices (dense or sparse), the sparse block matrix does not implement it.

4.9 PART SUMMARY

[Chapter 2](#) contains a brief introduction into nonlinear least squares methods and their extensions, the problems those methods are applicable or have been traditionally applied to and some of the state of the art solvers. A strong focus is on data structures: the problems discussed here are all sparse, with non-trivial sparsity patterns and the choice of the data representation affects the algorithms and ultimately the efficiency of the solution. [Chapter 3](#) discusses the standard formats for storing and manipulating sparse matrices.

Sparse matrices are often used in the implementations described in this chapter, as representing the problems by dense matrices would bring significant computational overhead and would quickly become impractical. Standard libraries for elementwise sparse matrices are popular, with Tim Davis' SuiteSparse being used notably often (10 out of 24 projects in the OpenSLAM² repository use it, Google's Ceres solver does as well).

This might well be due to the simplicity of its interface. E.g. rather than calling it by its [BLAS](#) designation `dcs_gemm`, the matrix-matrix multiplication routine is called `cs_multiply` in `Csparse`. Also, rather than passing each parameter and array comprising a sparse matrix separately (e.g. such function in Intel MKL [59] requires 15

² <http://openslam.org> – a platform for publishing [SLAM](#) implementations

arguments), it wraps sparse matrices in an easy to use structure `cs` (and thus only requires two arguments – the matrices to be multiplied).

Basic forms of sparse block matrices are used in the existing solvers, although they are practically always eventually being converted to elementwise sparse matrices for solving, see [Table 4.1](#). Standard formats for sparse block matrices are not being used and the authors of `NLS` solvers keep designing custom ones. NIST Sparse `BLAS` contains implementations of the routines for VBR format, but to our best knowledge, it is not being used in any of such solvers in robotics or computer vision. Admittedly, there is no standard `LAPACK` library for the VBR format which would provide matrix factorizations but iterative solvers would still be possible. Intel MKL does support the BSR format but that is of limited use as it would only allow solving problems where all the variables have the same dimension.

The common design goals in sparse block storage are:

DENSE BLOCKS: the blocks are stored as dense matrices so that they can be easily written by the Jacobian function rather than scattered into an elementwise format. This removes the bottleneck of matrix assembly.

IMMUTABLE ADDRESSES: in an incremental setting, the Jacobian matrix is augmented with new blocks as new observations come in and it can double as a cache if the addresses of the existing blocks do not change.

INTEGRAL REPRESENTATION: the nonlinear solvers often employ direct methods and matrix factorizations which are not compatible with split matrix schemes.

EFFICIENT ARITHMETICS: the current solvers, much to their disadvantage, only use sparse block matrices as a convenient platform for generation of elementwise sparse matrices to be passed to the linear solver.

In the following chapters, a *new sparse block matrix format* meeting this criteria will be introduced and compared to existing implementations. A suite of arithmetic routines comparable to `Csparse` in its extent is also supplied. Even though the proposed implementation is quite simple and not thoroughly tuned, it yields considerable performance and a simple `NLS` solver using this format easily outperforms the other state of the art solvers.

Part II

SLAM ++ THE SPARSE BLOCK MATRIX SOLVER

This part introduces a new sparse block matrix format proposed in this thesis. An efficient implementation of arithmetic routines for this new format is described as well. A simple nonlinear least squares solver, using this format at its core, is introduced and compared to the state of the art solvers. More efficient and novel methods for incremental solving and covariance recovery based on the block matrices are proposed as well.

SLAM ++ BLOCK MATRIX DESIGN

Many applications ranging from physics, computer graphics, computer vision to robotics rely on efficiently solving large nonlinear systems of equations, as illustrated in the previous chapter. In the case of using a Gauss-Newton-like algorithm, the solution can be approximated by iteratively solving a series of linearized problems. In some applications, the size of the system can be considerably large. The most computationally demanding part is to assemble and solve the linearized system at each iteration. This chapter shows solutions that exploit both, the block structure and the sparsity of the corresponding matrices and offers very efficient methods to manipulate, assemble and perform arithmetic operations on them.

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.

The block matrices can be more or less permissive as to the shape and placement of the dense blocks. From the algorithmic point of view, the blocks can be overlapping or non-overlapping and at the same time aligned or unaligned. Note that any of the first three combinations can be converted to the fourth – aligned, non-overlapping – by fragmenting the blocks as needed and summing up the remaining fully overlapping blocks, as illustrated in Figure 5.1. The only downside is that in some cases, the fragmentation can leave many 1×1 blocks behind or even yield an elementwise sparse matrix.

E.g. the BSR format only allows square, aligned and non-overlapping blocks which must all be the same size.

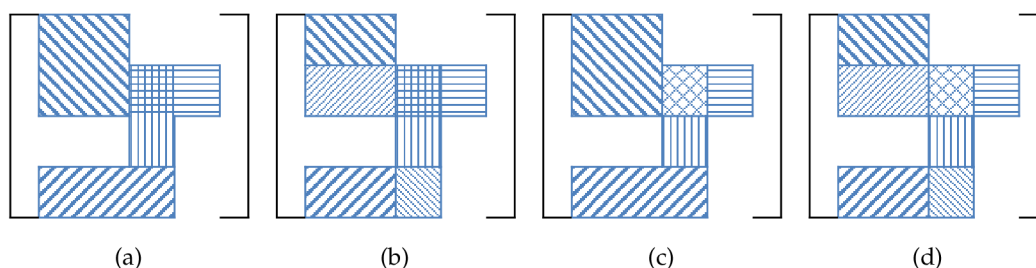


Figure 5.1: Block placements in sparse block matrices: a) unaligned block matrix with four blocks, two of which overlap, b) aligned block matrix – the unaligned blocks were fragmented (now there are 8 blocks two of which still overlap), c) unaligned with the overlapping blocks fragmented and fused (total of 5 blocks) and d) aligned non-overlapping block matrix (7 blocks).

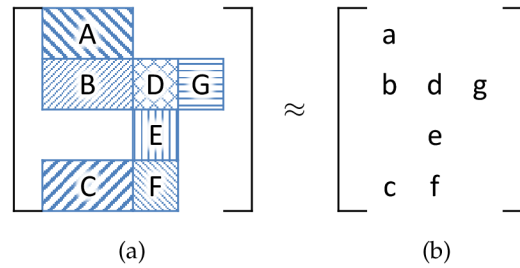


Figure 5.2: Relation of expressions on block and elementwise sparse matrices: a) aligned non-overlapping block matrix and b) a structurally equivalent elementwise sparse matrix.

An overlapping block matrix may be obtained e.g. by a procedure for finding block structure in general sparse matrices which aims at covering all matrix nonzeros by the minimum number of blocks possible, see e.g. [Figure 1.1c](#) or [Figure 5.1a](#). Unaligned block matrices ([Figure 1.1b](#) or [Figure 5.1c](#)) arise naturally e.g. in LOTs [30, 31] in image processing, where each two adjacent blocks overlap in order to avoid discontinuities in the processed image.

Assuming aligned, non-overlapping matrices has its benefits. Each block of the matrix can be treated as a (scalar) variable in an ordinary (elementwise) sparse matrix and formulas applicable to the elements can be automatically extended to blocks (see [Figure 5.2](#)), with the difference that scalar operations become operations on matrices: addition becomes elementwise addition of the blocks, multiplication becomes matrix multiplication, division becomes linear solving or backsubstitution in case the blocks are triangular, square root becomes Cholesky factorization. The only issue is that the blocks interacting in an arithmetic operation must have compatible dimensions. Fortunately, for most of the matrix algorithms, only the blocks in the same block row or block column are interacting and the dimensions are therefore guaranteed to match.

Similarly, operations taking multiple matrices as input (e.g. matrix addition or multiplication) can rely on the blocks of the two matrices to be aligned with each other. This makes the implementation of the arithmetic operations simpler and faster as only entire blocks interact (rather than the overlapping *parts* of the blocks interacting in case the matrices weren't aligned). In our implementation, it is required for the matrices to be aligned with each other, or in different words, to have a compatible block layout.

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 of 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 very 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 advantages of cache efficiency.

As seen in [Chapter 4](#), some of the existing state of the art NLS 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 e.g. CSparse [41] or Cholmod [42] libraries are used to perform the matrix factorization.

The advantage of *elementwise* sparse matrix schemes is that the arithmetic operations can be performed efficiently. Compressed sparse column (CSC) format [146] 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 sometimes also structural matrix modification, at the cost of slight memory overhead and reduced arithmetic efficiency, speed-wise.

Matrix assembly is a notable bottleneck in many situations: the time needed for putting the matrix together is comparable to the numerical operations which follow. The elementwise CSC representation [146] can be as efficient as any block matrix structure, in case of assembling a set of structurally-different matrices. The NLS 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.

In this chapter, a fast and cache efficient data structure for sparse block matrix representation is proposed, which combines the advantages of elementwise and block-wise schemes. It enables simple matrix modification, be it structural or numerical, while also maintaining, and often even exceeding the speed of elementwise operations schemes. Another important advantage of the proposed scheme is the overall robustness of the structure, allowing for validation and error-checking.

5.1 RELATED WORK

Many sparse linear algebra libraries are currently available. They range from implementations of basic arithmetics routines to complete linear algebra solutions [108, 46, 47, 8, 41, 73]. This chapter describes implementation and evaluation of matrix operations and storage, and it is particularly focused on matrices having a block structure. The operations we tackle are the building blocks for any least squares 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 [108, 46, 47],

containing simple operations on vectors and matrices, and [LAPACK](#) [8], 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 [41], developed by Tim Davis is one of the most used sparse linear algebra libraries in robotics and computer vision. 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 chapter. As mentioned above, CSparse stores its matrices in compressed column format which is suitable for operation on matrices, or in uncompressed coordinate format for simple matrix specification. Functions to convert between the formats are provided.

NIST Sparse [BLAS](#) [29] 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 implements two block matrix storage formats, constant block size (BSR) and variable block size (VBR) compressed sparse row and also their column-major alternatives (BSC and VBC). 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 difficult to use since the standard sparse block storage scheme is quite complex. To our best knowledge, it is the only library with [BLAS](#) interface to support the non-trivial VBR block matrices and one of a few to support them at all.

Sparskit [146] is a sparse matrix package written in Fortran. It has many matrix conversion routines, including conversions between CSR, BSR and VBR. It implements matrix-vector product for the VBR format and routines for finding block partitioning of a matrix. Interestingly, it does not implement routines for the simpler BSR format.

Intel Math Kernel Library (MKL) [59] is a closed-source dense and sparse [BLAS](#) and [LAPACK](#) implementation. It features C and Fortran interfaces and supports levels 2 and 3 functions for single block size (BSR) matrices. It also has other features such as Fast Fourier Transform, random number generation and data fitting.

PETsc [12] is library for scientific computation which includes support for sparse block matrices. It supports matrices in the single block size BSR format. It contains implementations of both direct and iterative solvers. It can use UMFPack [40] or Cholmod as back-ends.

As such, the BSR-only libraries are of limited use to applications with only single variable sizes. Sometimes it is possible to express a block matrix with different

block sizes as a sum of matrices with a single block size (often referred to as *splitting* [168, 139, 65]), e.g. for the CG solvers which only require matrix - vector product. For direct solvers, one option is to append the blocks with identity matrices in order for all the blocks to be of the same size but this naturally comes with performance penalties.

Buluç et al. [23] introduces a novel orientation-agnostic block matrix format (it is neither row- or column-major). It is called compressed sparse blocks (CSB) and works by partitioning the matrix to single size square blocks. The data of each block are represented in a coordinate format with reduced number of bits for row and column indices (since the blocks are small) and the data of all the blocks are serialized into three contiguous arrays. The blocks uniformly cover the entire matrix and one more array of pointers to the first element of each block is required. The blocks can therefore be accessed at random with the only disadvantage that completely zero blocks still have their pointer (although it points to an empty range of elements) so choosing very small block size becomes inefficient. The best performance was observed with block size of about 4096. Essentially while BSR stores a sparse collection of dense blocks, CSB stores dense collection of sparse blocks. The nonzero elements in each block should be ordered by Z-Morton ordering for better performance. While the serial performance of CSB is comparable to CSR, it gains higher performance rates in parallel processing, likely due to being more ordered and requiring lower bandwidth for the element indices.

Vuduc and Moon [175] describe a different kind of a block matrix format based on CSR. They call the new format unaligned block compressed sparse row (UBCSR). This format relaxes the alignment requirements of the BSR with the aim to reduce the zero fill-in caused by blocking. Additionally, they split the matrices with multiple block sizes. For conversion from CSR, the approximate block structure is first found and the matrix is converted to a VBR format. This is then split to several BSR matrices and finally the zero fill-in is reduced by un-aligning the blocks. This reduces both runtime and storage. The authors implement sparse matrix - vector multiply and gain about $2\times$ speedups compared to CSR.

g2o [106] (also see Section 4.6) is a library for solving NLS problems on graphical models. It contains an implementation of block matrix storage and supports a limited set of operations on it, essentially the matrix-vector and matrix-matrix products to be able to implement Schur and CG solvers. The block matrices in g2o are stored in block column-major order with the blocks of each column in a separate associative array (`std::map` in C++). This particular implementation of associative array guarantees immutable address of the blocks. g2o leverages this by storing pointers to the blocks in the corresponding graph factor objects (those generate the block values), eliminating repeated lookup. At the same time, `std::map` is implemented using a red-black tree, making the insertion of new blocks po-

tentially expensive and requiring multiple non-consecutive memory indirections upon lookup, increasing the likelihood of cache misses.

Ceres-solver [3] (also [Section 4.8](#)) is a library for solving [NLS](#) and regression problems. It 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 is not supposed to be employed by the users. The block matrices in Ceres are stored in a way, similar to the scheme described here¹, but their implementation does not allow for matrix modification and every time the block matrix changes structurally, it needs to be rebuilt. This is a major drawback for the iterative or incremental nonlinear solvers as a significant amount of time is lost in rebuilding the system matrix at every iteration. In one of the recent releases, Ceres adds support for split block matrices with the aim to handle problems with two types of variables, such as [BA](#) or [SfM](#), where the block sizes can be chosen from a prepared list of specializations². Those are however generated by an external Python script, making it somewhat difficult to use.

5.2 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 `g2o` block matrix implementation, the blocks are 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. In the Ceres implementation, the blocks are allocated in a linear array which would necessitate reallocation and data copying when incrementally adding new

¹ The representation described in here was developed independently, before Ceres was made public.

² This is not listed in the release notes as this functionality is hidden from the end user. Although no credit is given, this could well be an impact of one of our publications in this direction [[PŠI⁺13b](#), [PIS13a](#), [PIS⁺13b](#)].

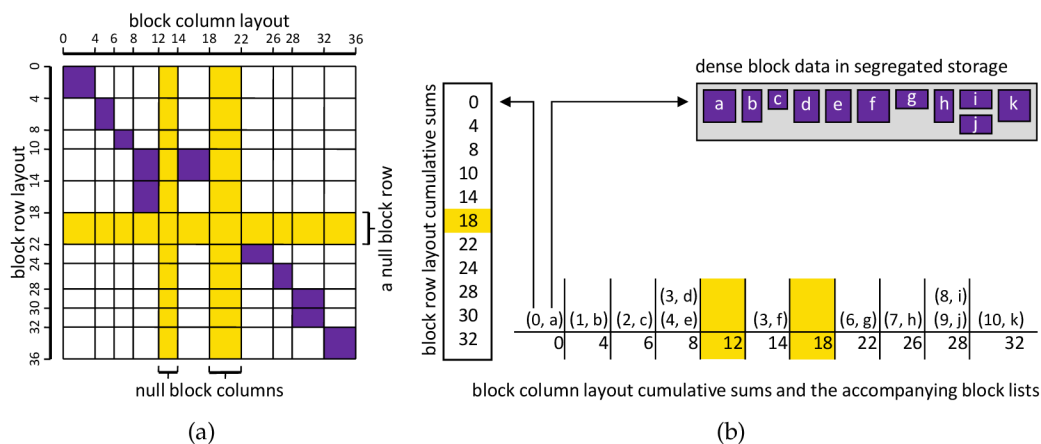


Figure 5.3: Block row / column layout of a block matrix. a) An example of a sparse block matrix and the actual values of the cumulative block sum (on top and left side). Non-zero dense blocks are shown in violet. Yellow shows null rows/columns. b) Dense block data in segregate storage. On the bottom, 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.

blocks to the matrix. It also uses element offsets rather than pointers, perhaps to avoid pointer arithmetics in reallocation but then pointer arithmetics is required every time when referencing the blocks. Additionally, Ceres does not align the memory, necessitating the use of slower unaligned [SIMD](#) instructions. To alleviate those problems, the proposed implementation allocates block memory in pages, which guarantees that the blocks are stored tightly next to each other while also allowing more blocks to be added without requiring to copy or shift the data.

The arithmetic efficiency of block matrices is mostly reduced, compared to elementwise sparse matrices, which might come as a surprise. That is because two or three extra inner loop counters for element rows and columns of the blocks are needed. This reduces the ratio of the arithmetics to flow control instructions.

Fortunately, in 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 therefore known in advance, at compile time. It is possible to use this information to hint the individual operations on matrices with lists of possible block sizes occurring in the operands. The proposed implementation is able to elegantly take advantage of this information using metaprogramming.

5.2.1 The Data Structure

In general, a vast majority of the existing block matrix schemes, including the proposed one, involves the same data layout as CSC representation (or an equivalent

one), but use more complex data structures to allow changes to the matrix structure which is useful especially in the context of incremental solving. For example, in the existing implementations [3, 105, 106] described in the previous section, trees or other higher abstract data types are used.

In the proposed block matrix implementation, block row and block column layouts are described using the same cumulative sum structure, as seen in [Figure 5.3a](#) on the top and left edge of the matrix. The columns structure additionally contains the lists of non-zero matrix blocks, each comprising of a row index and a pointer to matrix data.

Listing 5.1: The SLAM ++ sparse über block matrix format (C++; the constructors and member functions omitted).

```

1: class CUberBlockMatrix {
2: public:
3:     struct TRow { // a block row
4:         size_t n_height; // height of the row. in elements
5:         size_t n_cumulative_height_sum; // position of the block row
6:     };
7:
8:     struct TColumn { // a block column
9:         typedef std::pair<size_t, double*> TBlockEntry; // one block entry
10:
11:         size_t n_width; // width of the column. in elements
12:         size_t n_cumulative_width_sum; // position of the block column
13:         std::vector<TBlockEntry> block_list; // list of blocks
14:     };
15:
16:     typedef forward_allocated_pool<double, PAGE_SIZE,
17:         MEM_ALIGNMENT> _TyPool; // data storage type
18:
19:     // [constructors and operations on block matrices]
20:
21: protected:
22:     size_t m_n_row_num; // number of matrix rows, in elements
23:     size_t m_n_col_num; // number of matrix columns, in elements
24:     std::vector<TRow> m_block_rows_list; // list of sizes of blocks rows
25:     std::vector<TColumn> m_block_cols_list; // list of block columns
26:
27:     _TyPool m_data_pool; // data storage for matrix elements
28:     size_t m_n_ref_elem_num; // num. of referenced elements (shallow copy)
29: };

```

The elements themselves are stored in forward-allocated segregated storage (see [Figure 5.3b](#)), 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 coherence. The C++ declaration of the complete data structure can be seen in [Listing 5.1](#).

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, or near the end) but lookup is fast. At the same time the flat structure is cache friendly, allowing for fast iteration over the matrix data in arithmetics operations. 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 potentially incurred at every lookup. Also, traversal of all the nodes of the tree can be non-trivial.

To allow for the acceleration using vectorization by the [SIMD](#) instructions and to make hardware implementations easier, the blocks should be memory-aligned. E.g. for [SSE](#), the addresses of the first element of each block need to be an integer multiple of 64 bytes. Similarly, [GPUs](#) require so-called read coalescing which corresponds to alignment to 128 byte boundaries. It is possible in the proposed format to leave out unused entries so that each block is aligned (the pages are allocated aligned so that the first block is always aligned). In some cases, small blocks need not be aligned to save memory because vectorization would not be applied in such case (e.g. 1×1 blocks for [SSE](#)).

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 block rows and columns needs to be represented as well. Those are marked in yellow in [Figure 5.3a](#) and their representation is shown in [Figure 5.3b](#) where the fifth and sixth fields in the block column layout are empty and similarly the block row 5 is not referenced by any of the blocks.

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 $O(m_b + n_b + 2)$ 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 the structurally full-rank matrices in [NLS](#) problems there are no such null columns or rows, therefore, no extra space requirements apply.

5.2.2 Sparse Block Matrix Assembly

In order to write (scatter) a block into a matrix, the block column and block row need to be resolved first. Adding a new block row or column inside the matrix area, or alternatively 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 look a block up by its position given by element coordinates of the starting row and column, the block row and block column 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) 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 (2.7) or (2.11) every time a new linearization point is computed. 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 can operate with logical indexing.

The proposed implementation also allows for making shallow copies of matrices, where the block data is with the original matrix. That makes it possible to e.g. make permutation of a matrix using a fill-reducing ordering for factorization without the need to copy block data or to create triangular views. Any numerical modification to the original matrix is reflected in its copies. This feature is also vital in the context of nonlinear incremental solvers because it allows to reuse the permutation even after the linearization point (and so also the unordered matrix) has changed.

5.2.3 Basic Arithmetic Operations

The arithmetic operations on block matrices are typically carried out in the same manner as on elementwise 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

Algorithm 5.1: Naïve sparse block matrix multiplication.

```

1: function NAIVEMULT(A, B)
2:   C = NEWMATRIX(ROWS(A), COLS(B))
3:   for each columnBblock in B do
4:     colB = COLUMNOF(columnBblock)
5:     for each blockB in columnBblock do
6:       rowB = ROWOF(blockB)
7:       columnAblock = FINDCOLUMN(rowB, A) ▷ O(log nb)
8:       for each blockA in columnAblock do
9:         rowA = ROWOF(blockA)
10:        blockdest = FINDBLOCK(rowA, colB, C) ▷ O(log nb + log mb + log fb)
11:        blockdest = blockdest + blockA · blockB
12:      end for
13:    end for
14:  end for
15:  return C
16: end function

```

$O(\log n_b)$ lookup is used, and an example of the matrix multiplication is given in Algorithm 5.1. At line 7, an $O(\log n_b)$ lookup is required to find block column. Then at line 10, 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. This algorithm is otherwise efficient in the sense that each loop iteration calculates a single partial product and the number of iterations thus matches the number of Floating Point Operations (FLOPs) required by the matrix product at hand.

To improve performance, a function, mapping block rows of **B** to block columns of **A** can be used. Consider Algorithm 5.2: first, note the use of logical indexing of block rows and block columns by their id (lines 14 and 17), rather than by their physical position in elements which was used in Algorithm 5.1 (lines 7 and 10). This mapping is calculated as a projection from block rows of the **B** matrix to block columns of the **A** matrix using a modified ordered merge, as detailed in Algorithm 5.3 (a similar mapping is used also for matrix addition where it is calculated between row layouts and between column layouts of the matrices being added). The cost of calculating the mapping function is $O(m_b + n_b)$ 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. Note that the complexity involved is negligible, compared to the complexity of the arithmetic operation itself. This later allows to replace the logarithmic time lookup of $\text{columnA}_{\text{block}}$ by an $O(1)$ lookup. It also enables checking whether the matrix product is defined on the given block matrices.

Algorithm 5.2: Fast sparse block matrix multiplication.

```

1: function FASTMULT(A, B)
2:   C = NEWMATRIX(ROWS(A), COLS(B))
3:    $f_{\text{map}} = \text{BLOCKLAYOUTMAPPING}(\text{BLOCKCOLS}(\mathbf{A}), \text{BLOCKROWS}(\mathbf{B}))$ 
4:    $\text{colB}_{\text{id}} = 0$ 
5:   for each  $\text{columnB}_{\text{block}}$  in B do
6:     for each  $\text{blockB}$  in  $\text{columnB}_{\text{block}}$  do
7:        $\text{rowB}_{\text{id}} = \text{ROWIDOF}(\text{blockB})$ 
8:        $\text{columnA}_{\text{id}} = f_{\text{map}}(\text{rowB}_{\text{id}})$ 
9:       if  $\text{columnA}_{\text{id}} = \text{mismatch}$  then
10:        return  $\triangleright$  block layout mismatch, product not defined
11:       else if  $\text{columnA}_{\text{id}} = \text{null}$  then
12:        continue  $\triangleright$  the column in A is mismatched but also empty
13:       end if
14:        $\text{columnA}_{\text{block}} = \text{BLOCKCOLS}(\mathbf{A})[\text{columnA}_{\text{id}}]$   $\triangleright O(1)$ 
15:       for each  $\text{blockA}$  in  $\text{columnA}_{\text{block}}$  do
16:          $\text{rowA}_{\text{id}} = \text{ROWIDOF}(\text{blockA})$ 
17:          $\text{block}_{\text{dest}} = \text{FINDBLOCKLOG}(\text{rowA}_{\text{id}}, \text{colB}_{\text{id}}, \mathbf{C})$   $\triangleright \leq O(\log f_b)$ 
18:          $\text{block}_{\text{dest}} = \text{block}_{\text{dest}} + \text{blockA} \cdot \text{blockB}$ 
19:       end for
20:     end for
21:      $\text{colB}_{\text{id}} ++$ 
22:   end for
23:   return C
24: end function

```

Furthermore, insertion of a block only requires insertion into a sorted list which is up to $O(\log f_b)$ but avoids the lookup of block row and block column. For some types of operands (such as diagonal matrices or symmetric matrices), the order of the 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 $A^T A$ form. In the elementwise sparse matrix multiplication routines [75, 41], a helper dense vector is employed to accumulate the partial products and the nonzeros are then read out in linear time. This approach however produces matrices where the nonzeros in each column are not ordered by row. Sorting them would take $O(f_b \log f_b)$ time, which is equivalent to performing lookup f_b times in up to $O(\log f_b)$ time. The proposed algorithm is therefore not much slower.

As mentioned above, the block sizes correspond to the **DOF** of the variables and, in general, are known in advance. Using typelists [5] and templates, decision trees

Algorithm 5.3: Algorithm for Calculating Block Layout Mapping Function.

```

1: function BLOCKLAYOUTMAPPING(a, b)
2:    $m = \text{SIZE}(\mathbf{a}), n = \text{SIZE}(\mathbf{b})$ 
3:    $\mathbf{map}_a = \text{ZEROS}(1, m), \mathbf{map}_b = \text{ZEROS}(1, n)$ 
4:    $\text{count} = 0, \text{cum}_{\text{last}} = 0, \text{last}_a = 0, \text{last}_b = 0, i = 0, j = 0$ 
5:   while  $i < m \wedge j < n$  do ▷ merge the two layouts
6:      $\text{cum}_a = \text{BLOCKBASE}(\mathbf{a}[i]) + \text{BLOCKSIZE}(\mathbf{a}[i])$ 
7:      $\text{cum}_b = \text{BLOCKBASE}(\mathbf{b}[j]) + \text{BLOCKSIZE}(\mathbf{b}[j])$ 
8:      $\text{cum}_{\text{next}} = \min(\text{cum}_a, \text{cum}_b)$ 
9:     if  $\text{cum}_a = \text{cum}_{\text{next}}$  then
10:       $\mathbf{map}_a[i] = (\text{last}_a = \text{cum}_{\text{last}})? \text{count} : \text{mismatch}$ 
11:       $\text{last}_a = \text{cum}_{\text{next}}$ 
12:       $i++$ 
13:    end if
14:    if  $\text{cum}_b = \text{cum}_{\text{next}}$  then
15:       $\mathbf{map}_b[j] = (\text{last}_b = \text{cum}_{\text{last}})? \text{count} : \text{mismatch}$ 
16:       $\text{last}_b = \text{cum}_{\text{next}}$ 
17:       $j++$ 
18:    end if
19:     $\text{cum}_{\text{last}} = \text{cum}_{\text{next}}$ 
20:     $\text{count}++$ 
21:  end while

22:   $\mathbf{inv}_a = \text{REPMAT}(\text{null}, 1, \text{count})$  ▷ make a vector of count “null” symbols
23:  for  $i = 0$  to  $m - 1$  do
24:    if  $\mathbf{map}_a[i] \neq \text{mismatch}$  then
25:       $\mathbf{inv}_a[\mathbf{map}_a[i]] = i$  ▷ invert permutation  $\mathbf{map}_a$ 
26:    end if
27:  end for

28:   $\mathbf{f}_{\text{map}} = \text{ZEROS}(1, n)$ 
29:  for  $i = 0$  to  $n - 1$  do
30:     $\mathbf{f}_{\text{map}}[i] = (\mathbf{map}_b[i] \neq \text{mismatch})? \mathbf{inv}_a[\mathbf{map}_b[i]] : \text{mismatch}$ 
31:  end for ▷ compose permutations
32:  return  $\mathbf{f}_{\text{map}}$ 
33: end function

```

are built at compile time that later at runtime enable the use of dense kernels generated for a given block size. This allows for optimization using loop unrolling

and vectorization at the block level, e.g. in [Algorithm 5.2](#) at line 18. 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 the proposed C++ implementation, 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 certain types or instances of matrices to only contain blocks of specified sizes, but such solution was seen as less versatile, and was not implemented. The implementation details are described in [Appendix A](#).

5.2.4 Sparse Block Matrix Factorizations

An indispensable tool for solving linear systems, most of the matrix factorizations borrow from, is Gaussian elimination. Gaussian elimination modifies a matrix into its upper-triangular form by performing linear combinations of rows and at the same time modifies the right-hand side. The solution of a triangular system is easily found by backsubstitution: the last variable does not depend on any other and the solution is a simple ratio. The second last variable depends only on the last but now that it is known, it can be substituted to get a simple linear equation. The rest of the variables are solved for in similar manner, proceeding backwards, from the right to the left – hence the name back-substitution.

An important problem in Gaussian elimination (and most of matrix factorizations in general), is stability: the elimination involves division by the diagonal element (a *pivot*). If this division is by a small number, numerical issues ensue. A simple example might be the following matrix:

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 0 \end{pmatrix}. \quad (5.1)$$

Eliminating the 1 to get the matrix into the upper-triangular form requires division by a small quantity ϵ which will in turn amplify roundoff errors. A simple solution is to swap the rows first (and equally swap the rows of the right hand side). This process is called *pivoting*. The pivot can be chosen as the element of maximum magnitude, either only from the current column (partial pivoting) or from the lower-right submatrix that was not eliminated yet (full pivoting). Full pivoting is understandably slower but typically leads to more numerically robust algorithms.

One disadvantage of Gaussian elimination becomes apparent in solving multiple right-hand sides: although the right hand sides can be modified by the row operations simultaneously, a problem appears if not all the right hand sides are available at the same time. It is possible to gather the row operations in a matrix

instead which can later be used to multiply each right hand side and apply those operations to it. Enter matrix factorizations.

Cholesky factorization is a decomposition of a symmetric positive-definite matrix Λ to a product³ $R^\top R$. Matrices involved in normal equations of NLS are positive-definite and thus Cholesky factorization is a popular method. To solve a system of linear equations in the form $\Lambda x = b$, one first solves $R^\top y = b$ and then $Rx = y$ by forward- and back-substitution. Blocking Cholesky factorization is popular in *dense* linear algebra and is implemented e.g. in Eigen [73] but to our best knowledge, our sparse block Cholesky implementation is the first of its kind.

Due to symmetry, Cholesky factorization can be row-wise or column-wise. Additionally, the order of elimination can produce a row (a column) at a time (gather), or can modify the whole submatrix (scatter). These two cases are illustrated in Algorithm 5.4, functions COLUMNCHOL and SUBMATRIXCHOL (note that the RMOD function is a row-wise variant of CMOD from line 7 and was omitted to save space), respectively. Those modify the matrix and calculate the factorization in-place. An appealing property of Cholesky factorization is that no pivoting is needed.

In the sparse case, the order of operation is typically given by the underlying format. Since the proposed block format is derived from CSC, the COLUMN-CHOL is taken as the starting point. Implementing CDIV is trivial, with the exception that the loop is over nonzero blocks and the square root (line 2) becomes dense Cholesky decomposition and the division (line 4) becomes back-substitution with multiple right hand sides. Implementing CMOD involves some more trickery: the dot product of the two columns (line 9) becomes $\Lambda_{i,k}^\top \cdot \Lambda_{i,j}$ and needs to be resolved efficiently. Due to the sparsity, not all the columns will have blocks at the same positions so their contribution would be zero. Choosing the columns k that modify the current column j can be done efficiently using the elimination tree structure [41], a tree of variable dependences. To find the elements at the same row in the two columns, it is possible to employ a dense vector for the j^{th} column, in the style of CSparse. For the block case, this could cost quite a lot of additional storage therefore a different strategy using ordered merge (which runs in linear time) is employed.

In sparse decompositions, a different notion of blocking is sometimes used. In some cases, several consecutive columns in the factorization will have the same

³ Or alternatively as $\Lambda = LL^\top$ where $L \triangleq R^\top$. In this work, upper-triangular matrices are preferred, as most of the Cholesky factorization routines, including Cholmod, *read* only the upper-triangular part of the matrix and there seems to be some integrity in also *writing* an upper-triangular output. Additionally, for $A^\top A = \Lambda$ and $A = QR$ (where Q is orthogonal), it can be shown by writing $A^\top A = R^\top Q^\top QR = R^\top R$ that this R matrix is the same one as in the Cholesky factorization, up to the sign of the rows (Cholesky will always have positive diagonal entries). This choice of R over L is not motivated by any political or occult preferences.

Algorithm 5.4: Two Dense Cholesky Factorization Algorithms.

Require: That Λ is a symmetric, positive-definite $n \times n$ matrix

```

1: function CDIV(k,  $\Lambda$ )      ▷ Column k divided by the square root of the pivot.
2:    $\Lambda_{k,k} = \sqrt{\Lambda_{k,k}}$ 
3:   for i = 0 to k - 1 do
4:      $\Lambda_{i,k} = \Lambda_{i,k} / \Lambda_{k,k}$ 
5:   end for
6: end function

7: function CMOD(j, k,  $\Lambda$ )  ▷ Column j modified by a preceding column k (j > k).
8:   for i = 0 to k - 1 do
9:      $\Lambda_{k,j} = \Lambda_{k,j} - \Lambda_{i,k} \cdot \Lambda_{i,j}$       ▷ Gather (CMOD for L would scatter).
10:  end for
11: end function

12: function COLUMNCHOL( $\Lambda$ , n)
13:   for j = 0 to n - 1 do
14:     for k = 0 to j do                                     ▷ Left-looking, inclusive.
15:       CMOD(j, k,  $\Lambda$ )      ▷ Gather contributions of the preceding columns.
16:     end for
17:     CDIV(j,  $\Lambda$ )          ▷ Finalize the current column.
18:   end for
19: end function

20: function SUBMATRIXCHOL( $\Lambda$ , n)
21:   for k = 0 to n - 1 do
22:     CDIV(k,  $\Lambda$ )
23:     for j = k + 1 to n - 1 do                             ▷ Right-looking, exclusive.
24:       RMOD(j, k,  $\Lambda$ , n)      ▷ Scatter contributions from the current row.
25:     end for
26:   end for
27: end function

```

sparsity pattern, forming a dense block around the diagonal. This is commonly referred to as a *supernode*. While the proposed implementation and e.g. the one in CSparse are *simplicial*, Cholmod implements a supernodal factorization [33] which identifies these supernodes and uses dense kernels to speed the computation up. It would similarly be possible to identify block-supernodes in the block structure of the factorized matrix but its implementation was not attempted.

Another observation to be made about `CMOD` is that it can introduce new non-zero entries: for two columns j and k which have nonzero values in the same row above the diagonal, $R_{k,j}$ will be nonzero (ignoring possible numerical cancellation). This is commonly referred to as *fill-in*. The speed of the sparse factorization can be severely affected by the fill-in. A classical example is an arrow matrix:

$$\Lambda = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}, P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, P^T \Lambda P = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} \quad (5.2)$$

The Cholesky factorization of such Λ will be a full matrix (all the columns share nonzeros in the first row). However, an appropriate permutation P of the original system of equations can be employed, yielding no fill-in at all in the factorization of $P^T \Lambda P$. This requires the right-hand side vector (and the solution vector) to be permuted (inversely permuted) as well but that presents a negligible cost. Note that the permutation is rarely represented as a matrix in practice, rather it is represented as a vector of variable number reassignments (in this case $\mathbf{p} = (3, 0, 1, 2)$).

Finding the best fill-reducing permutation is an NP-complete problem [177], however many approximate algorithms are available. Based on the observation that the fill-in only occurs under the highest element of each column, initially the orderings strived to reduce the matrix profile or bandwidth, [66, 156, 58, 53], notably Reverse Cuthill-McKee (`RCM`) [37]. Later, orderings based on the elimination graph were proposed such as Exact Minimum Degree (`EMD`) and its modifications [112], Approximate Minimum Degree (`AMD`) [7] or Nested Dissection [64].

The ordering can be done on the level of elements (customary in sparse linear algebra) or on the level of blocks. The proposed implementation depends on the ordering of the block structure, otherwise the elementwise permutation could scatter the block structure completely. On the other hand, the block structure is represented by a much smaller matrix and the ordering heuristics thus run faster. At the same time, the quality of such ordering is comparable to the conventional one if not better [98].

5.2.5 Pivoting Sparse Block Matrix Factorizations

The major advantage of the Cholesky factorization is that it does not require pivoting – other factorizations are usually not numerically stable without one. This presents a serious issue in the context of sparse matrices if the pivoting is oblivious to the fill-in it causes, which in turn can present a significant explosion in space and complexity, as well as additional source of numerical problems. A related issue

in sparse block schemes is again the shattering of the block structure. This can be solved using threshold pivoting [152, 40] – a different pivot than the one proposed by the fill-reducing ordering is only used if it is of much larger magnitude, as defined by the threshold. Choosing a small threshold leads to pivoting for numerical stability while choosing a large threshold leads to pivoting for sparseness.

Threshold pivoting can still lead to large fill-in though. For instance, a Cholesky factorization of a $174,515 \times 174,515$ symmetric positive-definite matrix with 9,363,966 nonzero entries (a 3D reconstruction problem⁴) takes about 74 MB. However, LU factorization of the same matrix takes over 15 GB, due to less fortunate pivoting choices. This problem appeared when analyzing eigenvalues of the said matrix in R, using the rARPACK package⁵.

Applying the block structure to pivoting can lead to significant advantages, however. The pivot can only be chosen within the current block given by the fill-reducing ordering, which guarantees no changes in the sparsity pattern and thus no unexpected fill-in would occur. Since this can still lead to sub-optimal pivots in some rare cases, a different block can be chosen. Fortunately, to detect a sub-optimal pivot, only the diagonal block needs to be factorized, without modifying the off-diagonal entries, which presents a relatively small or even constant cost. If the factorization of the diagonal block fails, another block is chosen as the pivotal one, and at that point fill-in may occur. This fill-in is comparable to the one caused by the elementwise threshold pivoting algorithm. If no pivotal block is viable, the factorization could choose to either fail or to shatter the block structure and continue with elementwise pivoting.

5.3 PERFORMANCE ANALYSIS

In this section, the timing results for several matrix operations performed using the proposed implementation are compared to similar state of the art implementations such as CSparse, Ceres and NIST Sparse BLAS. NIST implementation can store matrices in several formats. CSR is a compressed sparse row elementwise 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 (as

⁴ The *Guildford Cathedral* dataset from <http://cvssp.org/impart/>

⁵ <http://cran.r-project.org/package=rARPACK>

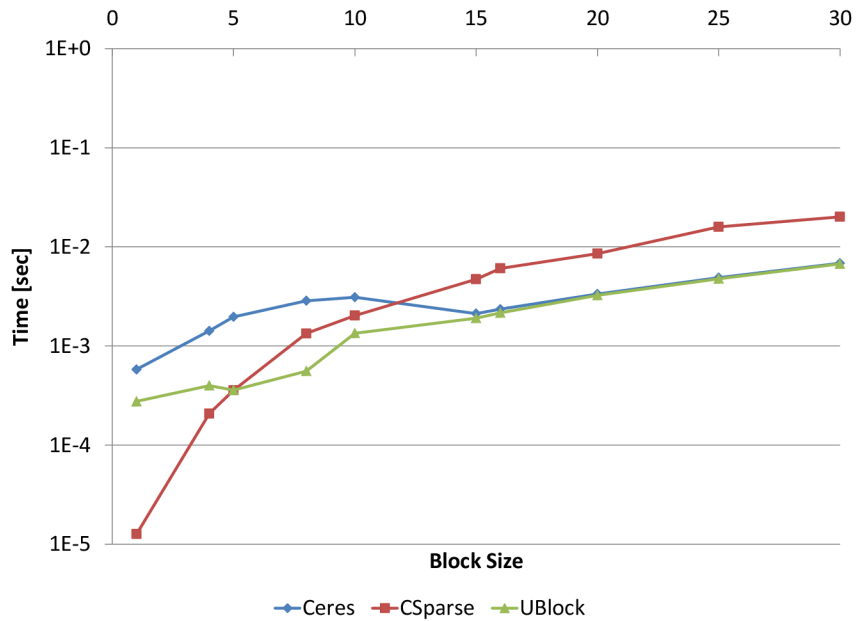


Figure 5.4: Time for compression of the MCCA matrix (smaller is better).

per [Listing 5.1](#)), and the version with metaprogramming optimization is denoted UBlock [FBS](#) (fixed block size) or “UB. [FBS](#)” for short.

All the tests were performed on a computer with Intel Core i5 [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.

The evaluation was performed on a subset of the The University of Florida Sparse Matrix Collection [[39](#)]. 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. Note that the goal of this benchmark was to ascertain the performance scaling and for that reason, only the structure of the matrices was used. In the tests, 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 naturally occurring block structure, such as (but not limited to) [NLS](#), [FEM](#) or [PDE](#).

Several matrices were selected for comparison. In particular, the MCCA matrix from the Harwell-Boeing [[52](#)] collection, 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

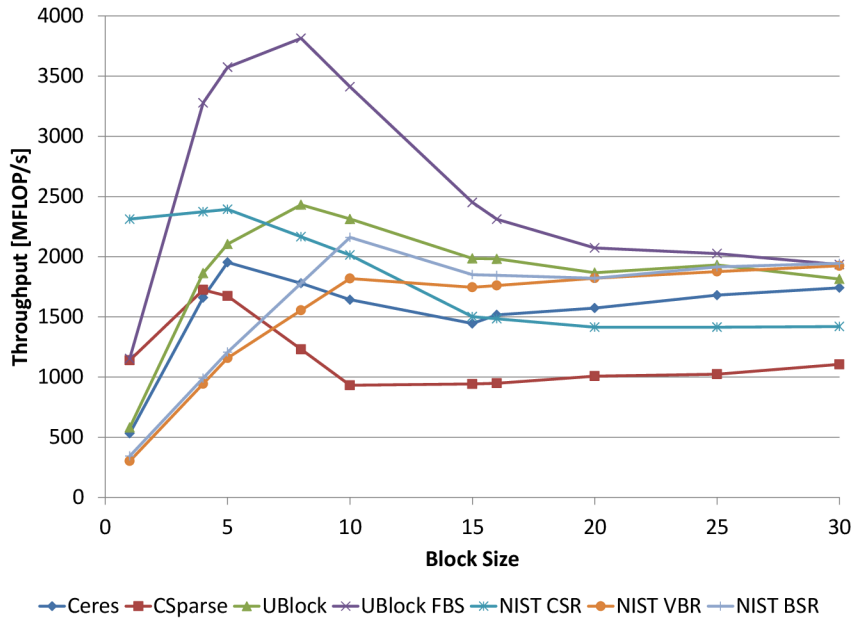


Figure 5.5: Performance scaling of general matrix vector product on the MCCA matrix.

experimental evaluation [29] on it. Since the NIST BLAS is not widely used, this limited comparison should be sufficient. For the rest of the evaluations, 200 matrices from The University of Florida Sparse Matrix Collection were chosen randomly.

A comparison of the time required to compress a sparse matrix using CSparse, Ceres and our implementation is shown in Figure 5.4. 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).

Similarly, Figure 5.5 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 elementwise implementation and the proposed fixed block size implementation. Although the NIST elementwise 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 elementwise 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 blocks and is the fastest for 8×8 blocks or larger. However, the proposed fixed block size implementation is always the fastest, except that the NIST CSR is faster for 1×1 blocks (but our implementation is still slightly ahead of the CSparse library).

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

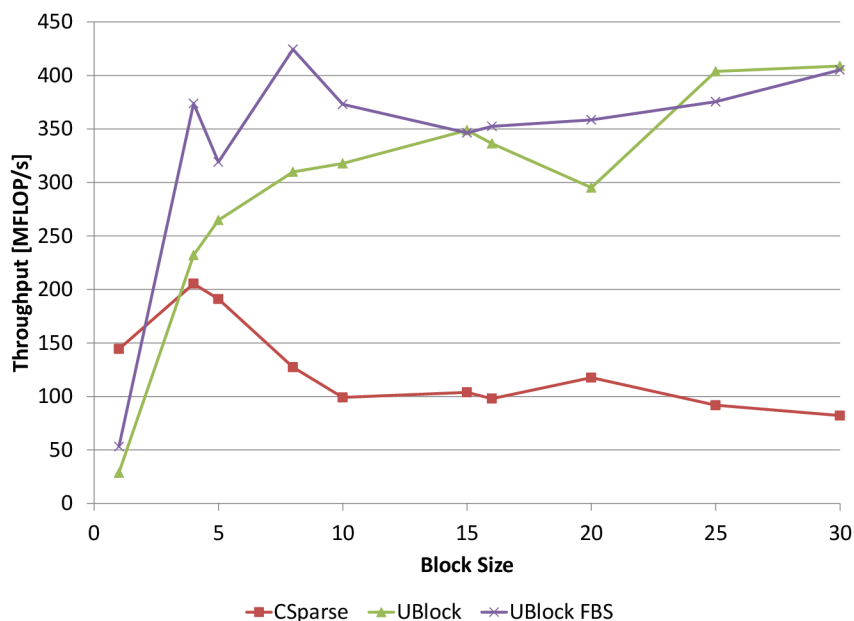


Figure 5.6: Performance scaling of linear combination of the MCCA matrix and its transpose (note that it is a square, nonsymmetric matrix).

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 5.6. 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 more optimized code for blocks of sizes that are multiples of four, since the SSE registers store four values.

Multiplication benchmark in Figure 5.7 displays similar behavior. Note that the gap between elementwise 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 . On the other hand, problems from the field of the computational chemistry may use even larger blocks. Still, it is fast enough to outperform even elementwise sparse implementations running on GPU, as will be demonstrated later on.

We also performed cache profiling using the Cachegrind⁶ 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. 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%.

⁶ A part of the Valgrind tool family, see <http://www.valgrind.org/info/tools.html#cachegrind>.

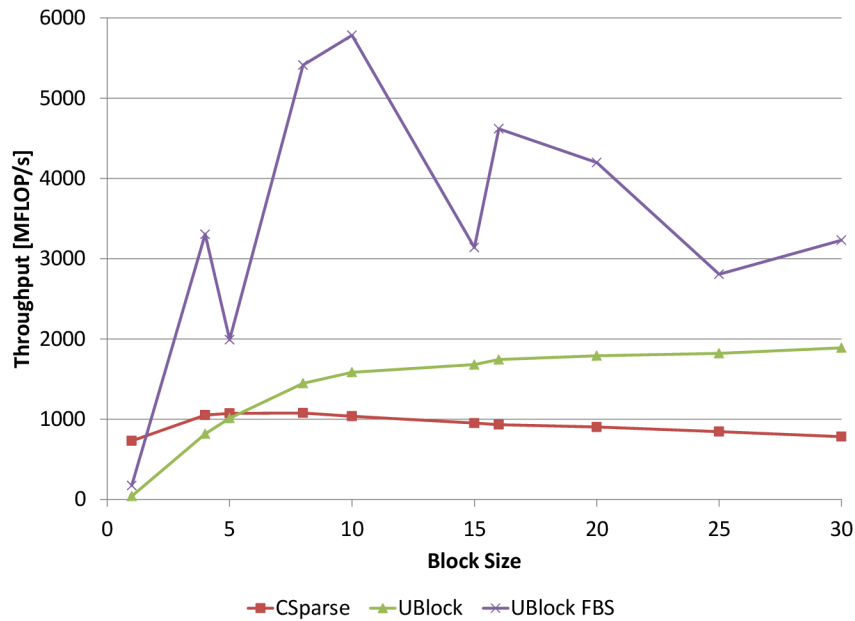


Figure 5.7: Performance scaling of the product of the MCCA matrix and its transpose.

Additional benchmarks are shown in [Table 5.1](#), which contains the average run times on 200 randomly chosen matrices from The University of Florida Sparse Matrix Collection [39]. 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. 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.

Cholesky factorization benchmarks are in [Table 5.2](#), which contains the average run times on 200 other randomly chosen “Cholesky candidate” matrices from The University of Florida Sparse Matrix Collection [39]. In here, the elementwise and blockwise factorizations used the same variable ordering (and therefore required the same amounts of FLOPs). The block size was varied from 1×1 to 6×6 (each block was initialized to the value of the original sparse matrix element and its

Table 5.1: Timing results on a subset of University of Florida Sparse Matrix Collection [39]; the best times are in bold.

		Block Size						
		1 × 1	4 × 4	5 × 5	8 × 8	10 × 10	15 × 15	16 × 16
Operation	Library	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
	UB. 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
	UB. FBS	3.330	8.440	20.506	31.895	55.363	261.459	242.498
$A^T \cdot A$ Product	UBlock	4.821	12.256	18.360	49.159	85.476	257.207	310.401
	UB. 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
	UB. 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
	UB. FBS	0.014	0.089	0.155	0.455	0.661	1.472	1.763

diagonal was multiplied by two to ensure the matrix stays positive definite rather than becoming semi-definite). The fixed block size version of our implementation is the fastest for 3×3 (which corresponds to 2D problems in robotics) or larger. The generic implementation requires larger blocks to be efficient and becomes faster than CSparse for 6×6 blocks.

In solving FEM problems and perhaps also in other methods which rely on highly efficient matrix vector products, an approach called *splitting* [168, 139, 65] can be employed. It refers to representing a matrix with blocks of multiple different sizes as a sum of several matrices, each containing blocks of one particular size. Then, each of those matrices can be represented using a simpler block matrix format and loops can be unrolled similarly as in the proposed Fixed Block Size (FBS) approach. To compare the performance of the splitting approach to the proposed decision tree

Table 5.2: Timing results of sparse block Cholesky factorization benchmark on a subset of University of Florida Sparse Matrix Collection [39]; the best times are in bold.

Benchmark	Library	Mode	Block Size					
			1×1	2×2	3×3	4×4	5×5	6×6
Cholesky	CSparse		1.233	1.850	0.765	2.065	3.034	1.166
	UBlock		14.910	5.601	1.244	2.617	3.314	1.084
		FBS	4.095	2.080	0.569	1.542	1.911	0.655

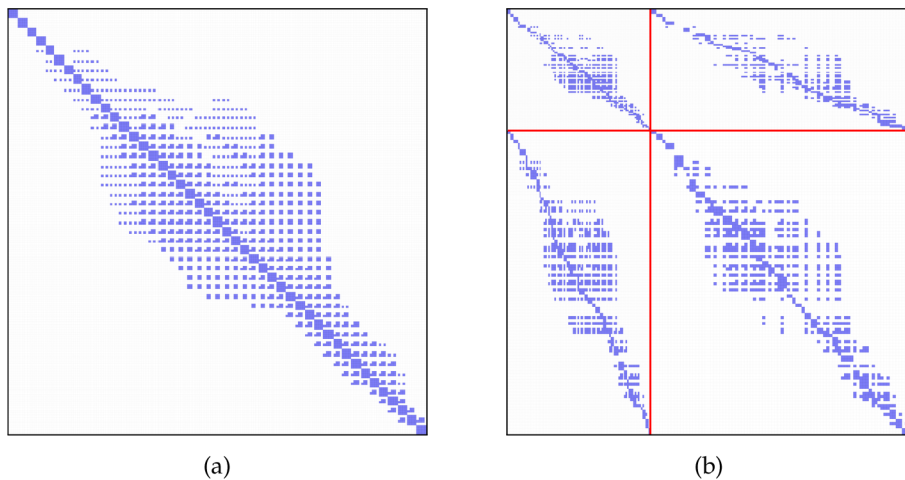


Figure 5.8: The MCCA matrix a) with the elements inflated to blocks of four different sizes and b) its split form.

approach, one more benchmark was performed. The MCCA matrix was used, and again its elements were inflated to blocks. In contrast to the previous benchmarks of performance scaling which used a single block size in the entire matrix, mixtures of *different* block sizes were generated. The mean block size was 9×9 for all cases, so that the number of FLOPs would be the same for all the tests. An example for four different block sizes is given in Figure 5.8. On the right, the matrix is reordered so that it can be split to four independent matrices, each of which contains only blocks of a single size. The matrix vector product is then performed separately for each of the four sub-matrices and the results are summed up.

The results for this benchmark are in Figure 5.9. It can be seen that CSparse has the same performance for all the tests, since it does not work with blocks at all. Similarly, NIST BLAS vBR and the proposed scheme denoted UBlock achieve relatively constant performance. Surprisingly, Ceres only achieves good performance for matrices with a single block size and then drops to the performance of CSparse and lower, even though it does not optimize for matrices with a single block size.

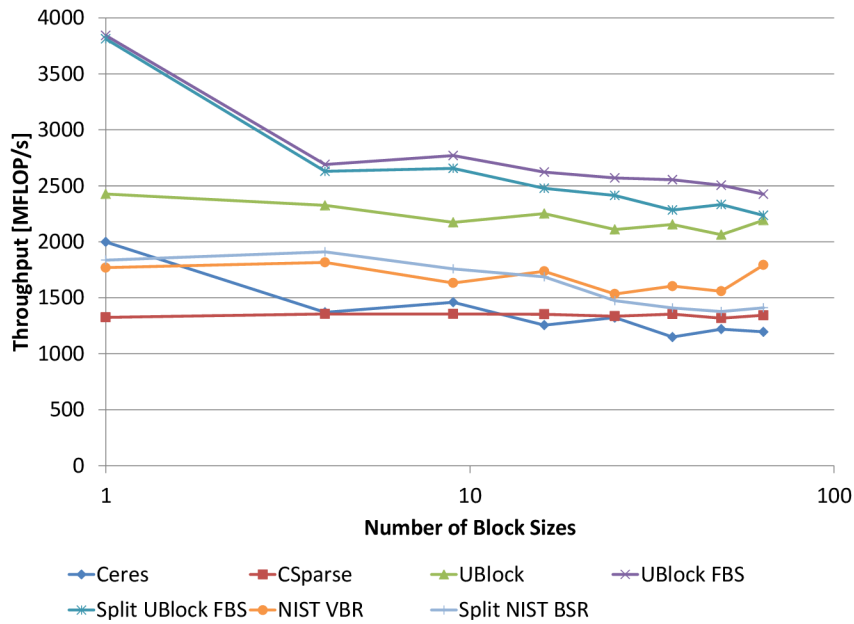


Figure 5.9: Comparison of splitting and the variable block size approaches.

The split approach implemented using the BSR format of the NIST BLAS, denoted Split NIST BSR, achieves slightly higher performance than the VBR format up to 9 block sizes, then it becomes slower. The small yield is given by this implementation not being able to unroll the loops. The version of the split approach implemented using the proposed block matrix scheme with the loops unrolled, denoted Split UBlock FBS, gains much higher performance and always stays ahead of the proposed variable block scheme, although for more than 64 block sizes, it would drop below. The decision tree approach using the non-split matrix denoted UBlock FBS achieves better performance than the split one, with the performance decreasing at lower rate with the growing number of block sizes. The performance hit of the decision tree version is related to the base 2 logarithm of the number of block sizes, while the performance hit of the splitting approach is related to the number of block sizes directly. On top of that, splitting also increases the bandwidth of the matrix, further increasing memory traffic. Note that for the splitting approaches, the time needed to reorder and split the matrix is not included in this evaluation.

The BSR format actually supports matrices with square blocks only. An extended rBSR that supports rectangular blocks was implemented in the same NIST style.

5.4 CHAPTER SUMMARY

A new implementation for block matrix operations was proposed in this chapter. It implements highly efficient kernels that are core for Nonlinear Least Squares (NLS) solvers. We targeted problems that have a particular block structure, where the size of the blocks corresponds to the number of Degrees of Freedom (DOFs) of the variables.

The proposed scheme combines the advantages of *block* schemes convenient in both, numeric and structural matrix modification and *elementwise*, 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, especially on the [GPU](#).

In the previous chapter, a fast implementation of operations on sparse block matrices was introduced and its performance was evaluated on more or less synthetic dataset obtained by “inflating” elementwise sparse matrices into block matrices. This chapter discusses design of an efficient nonlinear least squares solver based on the block matrices and evaluates its performance on several well-known [SLAM](#) problems. We refer to gathering all the constraints and variables and calculating the solution at once as *batch* solving. In contrast, *incremental* solving would be first solving a small part of the problem, then adding some variables and constraints, solving this larger problem again, and so on. This scenario typically arises in *online* robotic applications where a robot is traveling through the environment, gathering data and at the same time requiring estimates of its position and of the map before it can plan its next actions.

In robotics, Simultaneous Localization and Mapping ([SLAM](#)) is often formulated as a nonlinear least squares problem. Similar problems such as Structure from Motion ([SfM](#)) in computer vision [55] or elastodynamic simulations in computer graphics [81] rely on solving large nonlinear systems. Efficient incremental online algorithms for solving the underlying nonlinear least square problem are essential in real-time applications. Solving the nonlinear system is usually addressed by iteratively solving a sequence of linear systems (as described in [Section 2.1](#)). The most computationally demanding part is to assemble and solve the linearized system at each iteration.

The linear system can be solved either using direct or iterative methods. Direct methods, such as Cholesky or QR factorizations, are based on repeatedly factorizing a large matrix and backsubstitution to obtain the solution. Iterative methods, such as Conjugate Gradient ([CG](#)), on the other hand, employ matrix-vector multiplications and iteratively approximate the solution of the linear system. Iterative methods are more efficient from the storage (memory) point of view, since they only require access to the gradient, but they can suffer from poor convergence. Direct methods produce more accurate solutions and avoid convergence difficulties but they typically require a lot of storage as well as efficient elimination orderings to be found in order to maintain the sparsity of the resulting factors.

In robotics, approaching [SLAM](#) as a nonlinear optimization on graphs showed to provide very efficient solutions to moderate scale and well-behaved [SLAM](#) applications [45, 71, 95, 97, 106]. Graphs allow more natural representation of non-

linear least squares problems such as [SLAM](#), where a set of variables such as the robot poses and landmark positions are estimated, given a set of measurement constraints between those variables. The goal is to find the optimal configuration of the variables that maximally satisfy the set of nonlinear constraints. The existing methods repeatedly solve a sequence of linear systems in an iterative Gauss-Newton ([GN](#)) or Levenberg-Marquardt ([LM](#)) nonlinear solver. Real applications such as online mapping and localization of a robot in a large area and over very long period of time require extremely fast methods for building, updating and solving the sequence of linearized systems. It involves operating on matrices having a block structure, where the size of the blocks corresponds to the number of [DOF](#) of the variables.

Some of the existing implementations rely on sparse block-structure schemes [[105](#), [106](#)]. The block structure is maintained until the point of solving the linear system. Here is where [CSpase](#) [[41](#)] or [Cholmod](#) [[42](#)] libraries are used to perform the matrix factorization. Those are state of the art elementwise implementation of operations on sparse matrices.

6.1 RELATED WORK

This work focuses on the implementation of nonlinear least square solvers, involving direct methods. Several successful implementations of graph optimization techniques for [SLAM](#) already exist and have been used in robotic applications. In general, they are based on similar algorithmic framework, repeatedly applying Cholesky or QR factorizations in an iterative Gauss-Newton or Levenberg-Marquardt nonlinear solver. [g2o](#) [[106](#)] is an easy to use, open-source implementation which has been proven to be very fast in batch mode. It exploits the sparse connectivity and operates on the block-structure of the underlying graph problem. A similar scheme was initially implemented in [SSBA](#) [[105](#)] and [SPA](#) [[104](#)] and it is based on block-oriented sparse matrix manipulation. Using blocks is a natural way to minimize cache misses, since the [CPU](#) can automatically prefetch the data as they are accessed. Nevertheless, taking care about the layout of the individual blocks in the memory is very important, otherwise the overhead of handling the blocks can easily outweigh the advantage of cache efficiency.

However, in [SLAM](#) the state changes every step when new observations need to be integrated into the system. For very large problems, updating and solving every step can become very expensive. Incremental smoothing and mapping ([iSAM](#)) allows efficiently solving a nonlinear graph optimization problem in every step [[95](#)]. The implementation incrementally updates the R factor obtained from the QR factorization and performs backsubstitution to find the solution. The spar-

sity of the R factor is ensured by periodic reorderings. Recently, the Bayes tree data-structure [97, 98] was introduced to enable a better understanding of the link between sparse matrix factorization and inference in graphical models. The Bayes tree was applied to obtain iSAM2 [97, 98], which achieves high efficiency through incremental variable re-ordering and fluid relinearization, eliminating the need for periodic batch steps. When compared to the existing methods, iSAM2 performance finds a good balance between efficiency and accuracy. But still the complexity of maintaining the Bayes tree data structure can introduce several overheads.

The solutions proposed in this chapter aim to improve the above-mentioned implementations, which spend most of the time performing sparse matrix manipulation and arithmetic operations on sparse matrices. Our scheme is general, and can be easily incorporated into advanced incremental algorithms such as iSAM. Even iSAM2, which relies on a tree-like data structure, could benefit from the proposed scheme for the management of the dense blocks in memory.

6.2 INCREMENTAL SLAM

Online robotic applications require fast and accurate methods for the estimation of the current position of the robot. In an online application, the state is *incremented* with a new robot position and/or a new landmark every step and it is *updated* with the corresponding measurements. This translates into changing (2.10) by adding new block columns to the matrix A corresponding to each new variable (e.g. a pose or a landmark) and new block rows corresponding to each measurement [45]:

$$\hat{A} = \begin{pmatrix} A \\ A_u \end{pmatrix}, \hat{\mathbf{b}} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 + \mathbf{b}_u \end{pmatrix}, \quad (6.1)$$

where for the case of a single new measurement, $A_u = \mathbf{J}_k^\top \Sigma_k^{-1/2}$ and $\mathbf{b}_u = -\Sigma_k^{-1/2} \mathbf{r}_k$, with \mathbf{J}_k being the block row of the Jacobian matrix, corresponding to the residual \mathbf{r}_k of the measurement function $h_k(\theta_{i_k}, \theta_{j_k})$:

$$\mathbf{J}_k = \left(0 \dots \frac{\partial \mathbf{r}_k}{\partial \theta_{i_k}} \dots 0 \dots \frac{\partial \mathbf{r}_k}{\partial \theta_{j_k}} \right). \quad (6.2)$$

Note that the additions in (6.1) may require padding A with new zero columns and \mathbf{b}_2 with new zero rows in case new variables are added. Extension to multiple new measurements is trivial.

Similarly, for the Λ matrix in the normal equation (2.11), the increments translate to adding new block rows and block columns (as Λ is symmetric) with the size of

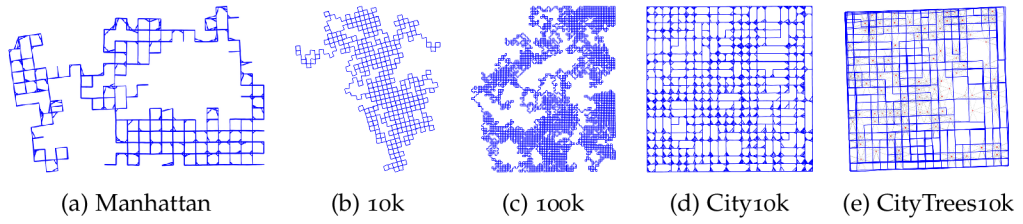


Figure 6.1: The synthetic datasets used in the batch solver evaluations.

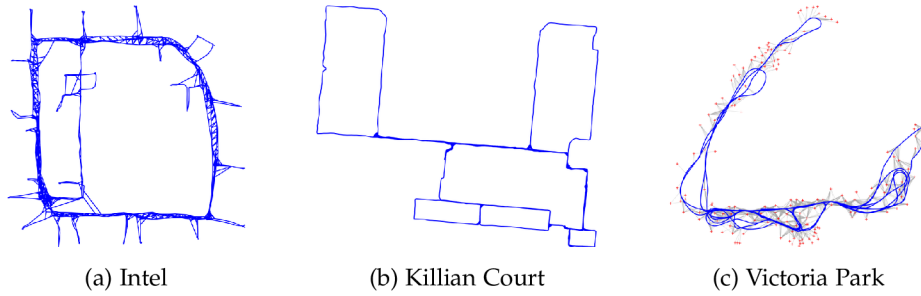


Figure 6.2: The real-world datasets used in the batch solver evaluations.

each new variable. Updates translate to (potentially) adding new nonzero entries. Updating Λ and η is additive:

$$\hat{\Lambda} = \begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^\top & \Lambda_{22} + \Omega \end{pmatrix}, \quad \hat{\eta} = \begin{pmatrix} \eta_1 \\ \eta_2 + \omega \end{pmatrix}, \quad (6.3)$$

where like for the A matrix above, Ω is the bottom-right section of $\mathbf{J}_k^\top \Sigma_k^{-1} \mathbf{J}_k$ and ω is the bottom part of $-\mathbf{J}_k \Sigma_k^{-1/2} \mathbf{r}_k$. Also, one can see that $\hat{\Lambda} = \Lambda + \mathbf{A}_u^\top \mathbf{A}_u$ and $\omega = \mathbf{J}_k \mathbf{b}_u$.

A *batch* computation of the solution of the new incremented and updated system is then performed at every n^{th} step. Ideally, the estimate is recalculated whenever new constraints or variables are added, to obtain the most accurate model of the environment that can be derived from all measurements gathered so far. For very large problems, batch solving at every step can become very expensive. Kaess et al [95, 97, 98] proposed efficient algorithms to incrementally solve the linear systems. Those algorithmic improvements offer very good solutions to online [SLAM](#) but they are out of scope of this chapter, which focuses on efficiently constructing the system at each iteration and speeding-up the basic arithmetic operations involved in batch solving.

6.3 IMPLEMENTATION DETAILS

In order to efficiently cope with very large nonlinear systems, the process of assembling and solving the sequence of linear systems must be as fast as possible. The

data structure has to allow for both, efficiently re-computing the values of the matrices A or Λ and the r.h.s. \mathbf{b} or $\boldsymbol{\eta}$ every time a new linearization point is available as well as efficiently updating the system when new measurements are available in incremental mode. One important characteristic of those matrices is their sparse block structure. For maintaining the Λ matrix, the individual Jacobian blocks J_k are cached and the data flow of the product $A^\top A$ is represented in such a way that it can be incrementally updated as the linearization point is changed.

Operating on dense blocks is a natural way to support vectorization and improve cache efficiency without any additional effort. Also, the division of the data in blocks allows efficient data representation at their natural granularity, making it simple to reference the data inside the matrix and change their value when needed.

6.4 EXPERIMENTAL EVALUATION

In order to evaluate our new efficient block matrix scheme, two standard graph SLAM algorithms were implemented; one that builds the linear system in (2.7), which is denoted allBatch- A and another one that increments the information matrix in (2.11), which is denoted allBatch- Λ . The timing results were compared to similar state of the art implementations such as iSAM [95], g2o [106], and SPA [104] (a 2D SLAM variant of sSBA [105]), which were described in further detail in Chapter 4. For SPA the svn revision 39478 of ROS (<http://www.ros.org/>) was used; for g2o, svn revision 29 from <http://openslam.org/> was used and for iSAM we used revision 7 from <https://svn.csail.mit.edu/isam>. Our implementation is available as open source at <http://sf.net/p/slam-plus-plus/>.

The implementations were evaluated on five standard simulated datasets; *Manhattan* [137], *10k* and *100k* [71], *City10k* and *CityTree10k* [94] and on three real datasets; *Intel* [85], *Killian Court* [21] and *Victoria Park* [133] (see Figure 6.1 and Figure 6.2). These are 2D SLAM datasets commonly used in evaluating graph-based SLAM implementations.

All the tests were performed on a computer with Intel Core i5 CPU 661 running at 3.33 GHz and 8 GB of RAM, the same machine as in the previous chapter. This is a quad-core CPU without hyperthreading and with full SSE instruction set support. Each test was run ten times and the average time was calculated in order to avoid measurement errors, especially on smaller datasets.

6.4.1 Tested Implementations

All the implementations used for comparisons are based on relatively similar algorithms, both in batch and incremental mode. Gauss-Newton non-linear solver was

Table 6.1: Time comparisons of the batch solvers (CM refers to Cholmod and CS refers to CSparse); the best times are in bold.

Dataset	CS		CM		CS		CM		χ^2 (iter.)
	g2o	iSAM	allBatch-A	allBatch- Λ	g2o	iSAM	allBatch-A	allBatch- Λ	
Manhattan	0.0614	0.0607	1.3641	0.0573	0.0613	0.0419	0.0468	6112.18 (5)	
10k	0.5539	0.5497	2.9518	0.6341	0.6977	0.4852	0.5798	171545.45 (6)	
100k	10.8135	9.4181	24.9582	10.4795	12.0097	9.2213	11.0566	8685.07 (6)	
City10k	0.4855	0.4491	1.4207	0.4635	0.5312	0.4203	0.4563	31931.41 (6)	
CityTrees10k	0.1359	0.1391	0.6245	0.1390	0.1469	0.0916	0.1090	548.50 (5)	
Intel	0.0066	0.0070	0.0356	0.0126	0.0083	0.0052	0.0060	559.05 (2)	
Killian Court	0.0084	0.0086	0.0535	0.0090	0.0095	0.0070	0.0075	$5 \cdot 10^{-6}$ (1)	

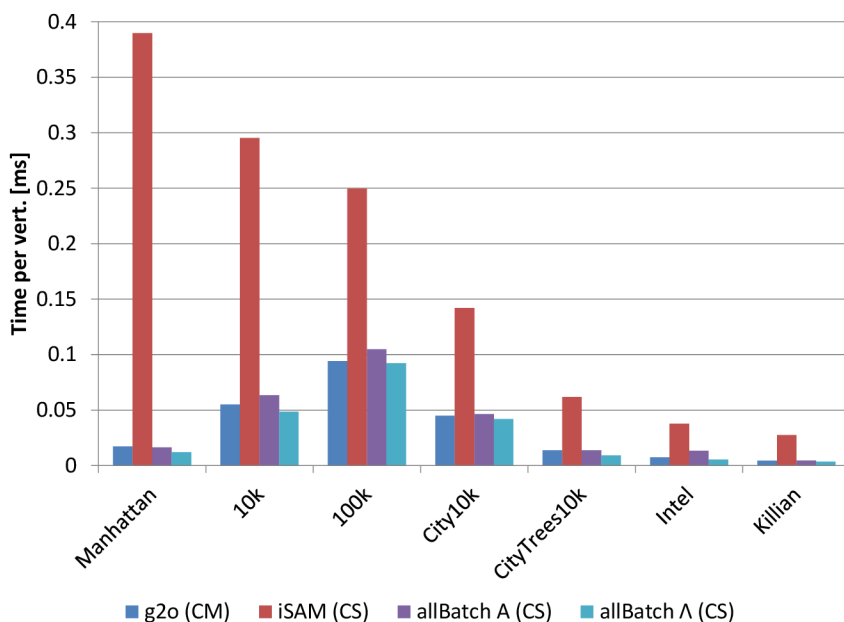


Figure 6.3: Comparisons of time *per vertex* in the batch solvers (CM refers to Cholmod and CS refers to CSparse).

tested in all cases, with the exception of SPA which uses Levenberg-Marquardt instead. iSAM has the possibility to perform incremental updates to solve at every step and to perform expensive batch steps only when needed, but for comparison purposes we tested only the cases where batch, update and solve are all done together.

g2o and SPA use their own sparse block matrix implementation. In g2o, it is based on a dense vector of trees, where each tree contains blocks for one column. This allows relatively fast random access to matrix elements, only $O(\log f)$ compared to $O(\log n_b + \log f)$ in our implementation. However, our implementation always avoids accessing blocks randomly, while in g2o this complexity is enforced on block lookup in matrix operations, making them slower than both CSparse and

our implementation. Overall, g2o is optimized for batch processing, but not for incremental solving.

The good SPA timings come from the fact that their implementation is optimized for the specific 2D pose adjustment problem (or bundle adjustment problem in case on sSBA), thus SPA is unable to process datasets with landmarks. In contrast, our implementation is general, allowing any combination of any block sizes.

The comparison with iSAM technically stands only for incremental every step. For incremental every 10 or every 100 steps, the other solvers perform state concatenation only and possibly also Jacobian computations. While the solution is still available at each step, the observation errors are only being reduced at every 10th or every 100th step, respectively. iSAM, on the other hand, is able to reduce this error in every step of the algorithm even between the 10th or 100th ones, using an approximate Gauss-Newton step which reuses the factorization from the previous linearization point (which is different from the current one – hence the approximation). But since the factorization takes most of the time in all the solvers, this comparison is still relevant.

6.4.2 Discussion of the Results

Timing results for running batch and incremental SLAM are shown in Tables 6.1 and 6.2 and Figures 6.3 and 6.4, respectively. Note that the accompanying figures show time per vertex, as it was hard to display the radically different times for all the datasets in a single plot. The *Victoria Park* dataset is not included in the batch tests since it does not converge if solved as batch. Similarly, the *100k* dataset is too large to be executed incrementally in reasonable time, and is not included either. The last column of Table 6.1 reports values of the χ^2 error and the number of iterations. Those are both the same (or very close in the case of χ^2) for all the tested solvers. The number of iterations was dictated by SPA, which does not allow setting the limit explicitly. In incremental mode, the tests were done using the linear solver which was the fastest in batch mode (Cholmod in case of g2o and CSparse in case of our implementation). The incremental results are split in three parts; solution updated every time a vertex is added, every 10 vertices and every 100 vertices.

Our implementation outperforms all the existing implementations in both batch and incremental mode. The comparison in batch mode shows a 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 such large speedups. Note that in this benchmark, the block Cholesky factorization is not used yet and so the proposed implemen-

Table 6.2: Time comparisons of the batch solvers running in the incremental mode (CM refers to Cholmod and CS refers to CSparse); the best times are in bold.

Dataset	g2o-CM	iSAM	SPA	allBatch-A-CS	allBatch-Λ-CS
Solve at each step					
Manhattan	94.9096	64.5844	23.8834	10.8883	10.0038
10k	2134.3000	1768.8400	515.2880	377.7490	329.1840
City10k	1326.6600	693.7860	308.0680	235.7910	222.5930
CityTrees10k	659.1590	434.7500	N/A	25.2809	22.7070
Intel	5.0513	4.4647	1.4763	0.8829	0.8424
Killian Court	20.8899	19.7519	5.6260	2.4275	2.1485
Victoria Park	293.1010	209.1740	N/A	30.6333	28.0194
Solve at each 10 steps					
Manhattan	9.5326	6.2510	2.5745	2.1462	1.9560
10k	211.2470	172.8720	62.6485	46.8314	42.0610
City10k	132.0070	68.5533	33.4328	28.8257	26.7019
CityTrees10k	65.0364	42.7519	N/A	13.2880	12.0940
Intel	0.5245	0.4541	0.1689	0.1336	0.1227
Killian Court	2.1518	1.9473	0.6392	0.3194	0.2794
Victoria Park	29.2946	20.7089	N/A	6.0668	5.5461
Solve at each 100 steps					
Manhattan	0.9891	0.6142	0.4446	0.3059	0.2853
10k	21.0767	17.0565	17.4968	6.2372	5.4294
City10k	13.3781	6.6846	5.4739	3.4363	3.0175
CityTrees10k	6.4883	4.1876	N/A	1.8136	1.5028
Intel	0.0695	0.0459	0.0371	0.0339	0.0292
Killian Court	0.2443	0.1915	0.1426	0.0904	0.0845
Victoria Park	2.9323	2.0580	N/A	0.8963	0.7522

tation also needs to resort to converting the block matrix to elementwise one and passing it to Cholmod or CSparse. The backsubstitution is then also performed using the elementwise code.

However, observe that there is some imbalance between small speedup in batch mode and large speedup in incremental mode. This stems from the simple fact that in batch, the system is only constructed once and most of the time is spent in the

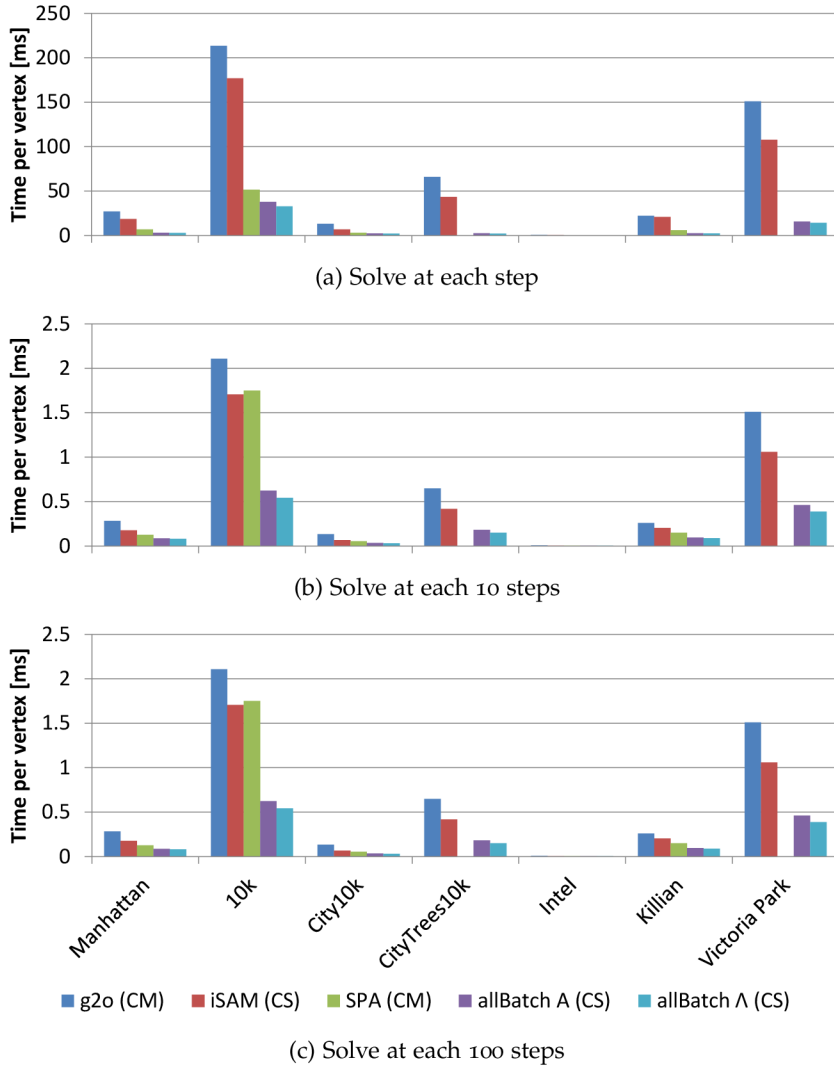


Figure 6.4: Comparisons of time *per vertex* in the batch solvers running in incremental mode (CM refers to Cholmod and CS refers to CSparse).

linear solver. In incremental mode, the block scheme starts paying off as more time is spent in building and updating the system matrix, especially on large datasets.

Due to the efficient block matrix operations described in Section 5.2, the difference between allBatch-A and allBatch- Λ is not very large, as updating Λ as in (6.3) with all the measurements is just an incremental version of the $A^T A$ product. Of course, when adding new variables and observations into the system, the upper-left submatrix of Λ doesn't change and in allBatch- Λ , this computation is saved. In allBatch-A, $A^T A$ must be calculated for the whole matrix, resulting in increased number of floating-point operations and slightly worse run times.

6.4.3 Block Operations Tests

Beyond the SLAM evaluation, matrix operations benchmarks were also ran on A and Λ matrices computed with the corresponding SLAM solution. Times for ele-

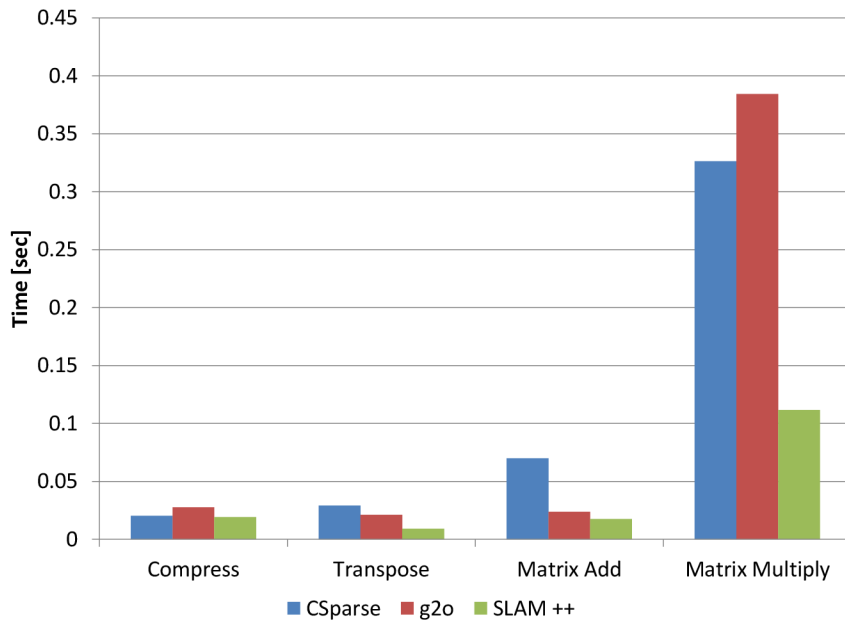


Figure 6.5: Time comparison of sparse block matrix operations performance on [SLAM](#) dataset matrices with 6×6 blocks. For the comparison with 3×3 blocks, please see [PŠI⁺13b].

mentary sparse matrix operations, such as *compression*, *transpose*, *addition* and *multiplication* were measured. Performance of CSparse [41], g2o [106] and our implementation were compared. SPA [104] was not included because its block matrix scheme is similar as in g2o. iSAM [95] was not included either, since it does not use any block matrix scheme. The results are shown in Figure 6.5.

Observe that CSparse is very good with matrix compression, since its data structure is the least complicated. But the compression must be performed every time the system is updated, making CSparse compression effectively slower after two iterations. In the other tests, our block matrix implementation outperforms CSparse. The most of the speedup comes from the use of vectorization. Furthermore, the block schemes prove to be more cache friendly than elementwise especially in the case of matrix transposition. In case of g2o [106], matrix transposition and multiplication is slower because of the use of the slow $O(\log f)$ block lookup, but those functions are not used in the optimization framework for [SLAM](#) (those would be used for [BA](#) or [SfM](#)).

6.5 CHAPTER SUMMARY

In [Chapter 5](#), a new implementation of sparse block matrix format and operations for it were proposed and individually benchmarked on a partially synthetic dataset. In this chapter, a basic implementation of a nonlinear least squares solver, operating in batch mode and using this new format, was proposed and compared

to other state of the art solvers. Simultaneous Localization and Mapping (SLAM) was chosen as a suitable application, since SLAM has a particular block structure, where the size of the blocks corresponds to the number of DOF of the variables. At the same time, it is relatively sparse and tractable using the unextended Gauss-Newton algorithm.

The proposed scheme combines the advantages of *block-wise* schemes convenient in both, numeric and structural matrix modification and *elementwise*, which allows efficient arithmetic operation. The advantage of the new scheme was demonstrated through an exhaustive comparison with the existing implementations in SLAM, on several publicly available datasets.

Even though the proposed scheme proved to significantly outperform the state of the art implementations in incremental mode, several improvements from algorithmic point of view can be applied. As already mentioned above, at this point it is just a batch solver operating in incremental mode. Performing incremental updates directly on the Cholesky factor, applying better ordering strategies (ordering is important to reduce the fill-in), and changing only the blocks corresponding to the affected variables should bring larger advantages.

The previous chapter discussed efficient methods for batch solving and although it touched the topic of incremental solving briefly, the implementation there did not really perform any increments and merely resorted to doing many batch steps of increasing size. While already quite fast, such approach is not very efficient as a lot of the computation is repeated unnecessarily. That is where the real *incremental* methods come in. Same as there, the focus of this chapter will be on solving SLAM problems efficiently but also precisely.

The challenge appears in online applications, where the state changes every step. In an online SLAM application, for example, every step the state is *incremented* with a new robot pose and with positions of the newly observed landmarks and it is *updated* with the corresponding measurements. For very large problems, updating and solving the nonlinear system at every step can become very expensive. Every iteration of the nonlinear solver involves building a new linear system using the current linearization point, calculating its factorization and solving. In here, calculating the factorization is typically the most expensive step.

This can be alleviated by changing the linearization point less frequently so that the factorization is not needed at every step. New variables can be added to the factorization e.g. using so called rank 1 updates [41, 33]. The solution to the linearized system can then be calculated at any time, using back-substitution (which runs at a fraction of time needed for the factorization). Although the Jacobian matrix (and so the linearization point) does not correspond to the state, approximate Gauss-Newton steps can still reduce the error, unless close to an abrupt change in the derivatives (such as in the vicinity of a singularity). This is essentially the iSAM algorithm [95], although it uses Q-less QR factorization rather than the Cholesky decomposition. It was later reimplemented in an experimental branch of g2o¹ using Cholmod's rank updates, with comparable results.

It would seem that the solution is to incrementally update the linear system in the already factorized form and to perform backsubstitution to compute the solution. However, there is still one more problem – the fill-in. Merely updating the factorization with new variables without ever applying a fill-reducing ordering would quickly lead to a massive fill-in ... and a correspondingly massive slowdown. In the context of robotics, this happens notoriously with so called *loop clo-*

¹ Can be found at <https://github.com/RainerKuemmerle/g2o>.

*sure*s which occur when the robot is returning to a place it has visited before and begins establishing links between the latest pose and some of the much older ones. In the matrix form, those links (measurements, observations) typically occupy far off-diagonal entries under which fill-in occurs.

Conversely, odometric measurements (the other prominent type of measurement in robotics; no matter whether measured using an odometry sensor, expected from the control commands to the actuators or calculated e.g. by laser scan matching) are between the consecutive poses only and can thus be handled relatively easily.

Unfortunately, there is no viable algorithm for performing matrix permutation once it has been factorized as of yet, so the authors of iSAM [95] settled for periodic reordering and batch re-factorization. On the other hand, different data structures were developed later that allow variable reordering in the factorization [98], so clearly it can be done also in matrices. This is typically done every 10 or every 100 steps in order to compromise between the fill-in rising uncontrollably and between performing too many batch steps.

The new method introduced in this chapter has the advantage that it adapts to the size of the updates and performs batch steps only when needed while still keeping the option to set the frequency of the batch steps. It is based on several optimizations of the incremental algorithm. The proposed implementation a) selects between three types of updates, depending on the size of the the update and the error b) uses double-constrained ordering by blocks c) performs backsubstitution by blocks and d) uses efficient block-matrix scheme for storage and arithmetic operations. These optimizations allow for very fast online execution of the algorithm and provide very accurate solutions at every step.

7.1 RELATED WORK

Several successful implementations of nonlinear least squares optimization techniques for SLAM already exist and have been used in robotic applications. In general, they are based on similar algorithmic framework, repeatedly applying Cholesky or QR factorizations in an iterative Gauss-Newton or Levenberg-Marquardt nonlinear solver. g2o [106] is an easy to use, open-source implementation which has been proven to be very fast in batch mode. It exploits the sparse connectivity and operates on the block-structure of the underlying graph problem.

A similar scheme was initially implemented in sSBA [105] and SPA [104] and it is based on block-oriented sparse matrix manipulation. Using blocks is a natural way to optimize the storage, nevertheless, taking care about the layout of the individual blocks in the memory is very important, otherwise the overhead of handling the blocks can easily outweigh the advantage of cache efficiency.

However, in SLAM the state changes at every step since new observations need to be integrated into the system. For very large problems, updating and solving every step can become very expensive. Incremental smoothing and mapping (iSAM) allows efficiently solving a nonlinear optimization problem in every step [95]. The implementation incrementally updates the R factor obtained from the QR factorization and performs backsubstitution to find the solution. To reduce the rank of those updates, the linearization point is only changed every 100 steps. The sparsity of the R factor is ensured by reordering upon relinearization.

Recently, the Bayes tree data-structure [96, 97, 98] was introduced to enable a better understanding of the link between sparse matrix factorization and inference in graphical models. The Bayes tree was used to obtain iSAM2 [97, 98], which achieves high efficiency through incremental variable reordering, eliminating the need for periodic batch steps, and through fluid relinearization. It is achieved by thresholding the update δ on a per-variable basis and updating only the significantly changing variables (the default threshold is 0.1). Similarly to iSAM, the linearization points are only changed every 10 steps. When compared to the existing methods, iSAM2 performance finds a good balance between efficiency and accuracy. But still the complexity of maintaining the Bayes tree data structure can introduce several overheads.

7.2 INCREMENTAL SLAM

The system in (2.7) can be incrementally built by appending the matrix Λ with new columns corresponding to each new variable (pose/landmark) and new rows corresponding to each measurement. We now shall focus more closely on the sparsity patterns involved – for each new measurement, the new block row is sparse and the only nonzero elements correspond to the Jacobians of the new residual.

For the normal equation in (2.11), the size of the matrix increments in number of rows and columns with the size of each new variable and it is updated by adding the new information to Λ and η . To match with the formulation in Section 2.1, and building on Section 6.2, the update step is:

$$\hat{\Lambda} = \Lambda + \begin{pmatrix} 0 & 0 \\ 0 & \Omega \end{pmatrix}, \hat{\eta} = \eta + \begin{pmatrix} 0 \\ \omega \end{pmatrix}. \quad (7.1)$$

Assuming that this update corresponds to adding a single new observation of the form $z_k = h_k(\theta_{i_k}, \theta_{j_k}) - v_k$ (regardless of whether it adds a new variable or not and without the loss of generality – extending to multiple new observations or measurements involving more than two variables is trivial), $\Omega = \tilde{\mathbf{J}}_k^T \Sigma_k^{-1} \tilde{\mathbf{J}}_k$ and

$\omega = -\tilde{\mathbf{J}}_k \Sigma_k^{-1/2} \mathbf{r}_k$, where $\tilde{\mathbf{J}}_k$ is the following block row of the Jacobian matrix, truncated so as to contain no zeros on the left:

$$\tilde{\mathbf{J}}_k = \mathbf{J}_{k_{\min(i_k, j_k):\text{end}}} = \begin{pmatrix} \frac{\partial \mathbf{r}_k}{\partial \theta_{i_k}} & \dots & 0 & \dots & \frac{\partial \mathbf{r}_k}{\partial \theta_{j_k}} \end{pmatrix}. \quad (7.2)$$

The derivatives of the residual function $\mathbf{r}_k = \mathbf{h}_k(\theta_{i_k}, \theta_{j_k}) \ominus \mathbf{z}_k$ with respect to its state variables θ_{i_k} and θ_{j_k} are referred to² as \mathbf{J}_{ki_k} and \mathbf{J}_{kj_k} below. The sparsity and the size of the Ω matrix are important for the incremental updates of the system. For the two affected variables, Ω will have four nonzero blocks:

$$\Omega = \begin{pmatrix} \mathbf{J}_{ki_k}^\top \Sigma_k^{-1} \mathbf{J}_{ki_k} & \dots & 0 & \dots & \mathbf{J}_{ki_k}^\top \Sigma_k^{-1} \mathbf{J}_{kj_k} \\ \vdots & & & & \vdots \\ 0 & & \ddots & & 0 \\ \vdots & & & & \vdots \\ \mathbf{J}_{kj_k}^\top \Sigma_k^{-1} \mathbf{J}_{ki_k} & \dots & 0 & \dots & \mathbf{J}_{kj_k}^\top \Sigma_k^{-1} \mathbf{J}_{kj_k} \end{pmatrix}. \quad (7.3)$$

Let us recall that the solution of the next Gauss-Newton step becomes $\hat{\Lambda} \delta = \hat{\eta}$ which can be obtained by calculating Cholesky factorization $\hat{\mathbf{R}}^\top \hat{\mathbf{R}} = \hat{\Lambda}$ and solving $\hat{\mathbf{R}}^\top \hat{\mathbf{d}} = \hat{\eta}$ and $\hat{\mathbf{R}} \delta = \hat{\mathbf{d}}$ using back and forward substitution. For very large problems, recalculating $\hat{\mathbf{R}}$ at every step becomes very expensive.

7.3 ALGEBRAIC INCREMENTAL UPDATES OF THE CHOLESKY FACTOR

In this section, the update of the Cholesky factor $\mathbf{R} \triangleq \text{chol}(\mathbf{R}^\top \mathbf{R})$ is discussed. This update is referred to as an *algebraic* one because it is slightly different from the rank update. It can be used in order to avoid unnecessary and expensive matrix factorizations every step. Observe that in (7.1) only a part of the information matrix and the information vector is changed in the update process and the same happens with the upper triangular factor \mathbf{R} . The updated $\hat{\mathbf{R}}$ factor and the corresponding r.h.s. $\hat{\mathbf{d}}$ can be written as:

$$\hat{\mathbf{R}} = \begin{pmatrix} \mathbf{R}_{11} & \mathbf{R}_{12} \\ 0 & \hat{\mathbf{R}}_{22} \end{pmatrix}, \quad \hat{\mathbf{d}} = \begin{pmatrix} \mathbf{d}_1 \\ \hat{\mathbf{d}}_2 \end{pmatrix}. \quad (7.4)$$

From $\hat{\Lambda} = \hat{\mathbf{R}}^\top \hat{\mathbf{R}}$ and (7.1), the equation (7.4) becomes:

$$\hat{\Lambda}_{22} = \Lambda_{22} + \Omega = \mathbf{R}_{12}^\top \mathbf{R}_{12} + \hat{\mathbf{R}}_{22}^\top \hat{\mathbf{R}}_{22}, \quad (7.5)$$

²Note that here, \mathbf{J}_{ki_k} and \mathbf{J}_{kj_k} are logical blocks of the matrix \mathbf{J} at block row k and block columns i_k and j_k , respectively, which correspond to the first and the $|i_k - j_k|^{\text{th}}$ elements of the block vector $\tilde{\mathbf{J}}_k$. The former notation is preferred, in order to avoid nested subscripts and notation clutter.

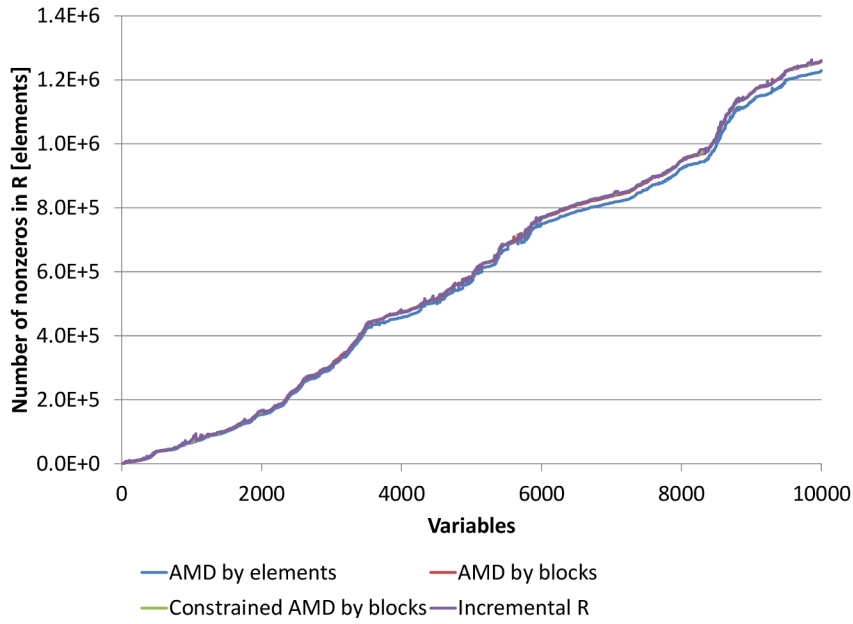


Figure 7.1: Evaluation of several ordering heuristics in terms of nonzero elements, compared to the actual number of non-zero elements in the Incremental R algorithm, on the *10k* dataset. Note that the results for orderings by block are practically identical.

and the part of the \hat{R} factor that changes after the update can be computed by applying Cholesky decomposition to this matrix of the same size as Ω :

$$\hat{R}_{22}^T \hat{R}_{22} = \Lambda_{22} + \Omega - R_{12}^T R_{12}, \quad (7.6)$$

$$\hat{R}_{22} = \text{chol}(\hat{\Lambda}_{22} - R_{12}^T R_{12}) \quad (7.7)$$

$$= \text{chol}(R_{22}^T R_{22} + \Omega). \quad (7.8)$$

Further in this chapter, (7.7) is referred to as *lambda*-update because it uses parts of the $\hat{\Lambda}$ to update R and similarly (7.8) is referred to as *omega*-update since it directly uses Ω to update R.

The part of the r.h.s. vector affected by the new measurement can also be easily updated. By expanding $\hat{R}^T \hat{\mathbf{d}} = \hat{\boldsymbol{\eta}}$ and focusing on the lower part that is changing, $\hat{\boldsymbol{\eta}}_2 = \boldsymbol{\eta}_2 + \boldsymbol{\omega} = R_{12}^T \mathbf{d}_1 + \hat{R}_{22}^T \mathbf{d}_2$ and so:

$$\hat{R}_{22}^T \hat{\mathbf{d}}_2 = \boldsymbol{\eta}_2 + \boldsymbol{\omega} - R_{12}^T \mathbf{d}_1, \quad (7.9)$$

$$\hat{\mathbf{d}}_2 = \hat{R}_{22}^T \setminus (\hat{\boldsymbol{\eta}}_2 - R_{12}^T \mathbf{d}_1), \quad (7.10)$$

where \setminus is linear solving operator; with \hat{R}_{22}^T being lower triangular, it can be realized using backsubstitution. After obtaining both \hat{R} and $\hat{\mathbf{d}}$, forward substitution can be performed to find the solution of the linear system $\hat{R}\boldsymbol{\delta} = \hat{\mathbf{d}}$.

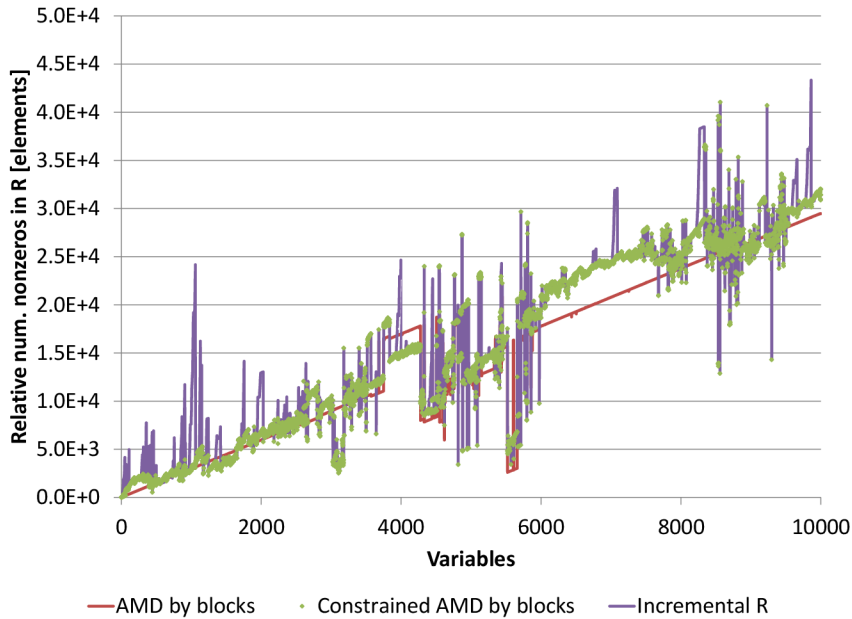


Figure 7.2: The fill-in relative to the best heuristic in Figure 7.1, which is AMD by elements. Please, note that the scale of this plot is about two orders of magnitude smaller than the absolute values of the fill-in. Figures for the *10k* dataset.

7.4 IMPLEMENTATION DETAILS

Online applications such as SLAM, require extremely fast methods for building, updating and solving the sequence of linearized systems. In this section, we introduce several optimizations towards high performance SLAM based on incremental updates of the factored representation.

7.4.1 Adaptive Updates

The proposed methodology adapts to the most favorable incremental update scheme, depending on the size of the updates. It considers three ways to update the system: 1) omega-updates, 2) lambda-updates and 3) updating the entire R, and applies heuristics to select the best strategy. Omega-updates in (7.8) are fast for small-size Ω because they involve the multiplication of small matrices $R_{22}^\top R_{22}$ which can however be relatively dense. Therefore, this is not suitable when Ω is obtained from measurements that are far apart (e.g. loop-closures). In this case lambda-updates in (7.7) are faster since they involve the multiplication of typically very sparse matrices $R_{12}^\top R_{12}$.

Updating very large loops becomes expensive due to bookkeeping. When loop length approaches the number of variables in the system, recalculating R by applying full Cholesky decomposition to the Λ matrix becomes more efficient. Full factorization is also beneficial due to the fact that the ordering heuristics are ap-

plied to the entire Λ , which considerably reduces the fill-in of the factor R and speeds up the backsubstitution in the subsequent solving steps.

7.4.2 Efficient Ordering Strategies

The fill-in of the factor R directly affects the speed of the backsubstitution and the updates. Its sparsity depends on the order of the rows and columns of the matrix Λ , called *variable ordering*. Unfortunately, finding an ordering which minimizes the fill-in of R is NP-complete. Therefore, heuristics have been proposed in the literature [6] to reduce the fill-in of the result of the matrix factorization. In the proposed implementation, the constrained AMD ordering is used, available as a part of SuiteSparse family of libraries [41].

In an incremental SLAM process, the new variable – either the next observed landmark or the next robot pose – is always linked to the current pose in the representation. In order to be able to perform efficient incremental updates on the Cholesky factor, the last pose is constrained to be ordered last. This especially helps when updating using odometric constraints between the consecutive poses in Pose-SLAM type problems. For landmark SLAM, one landmark is often observed from several poses. Without an additional constraint, a recently observed landmark can be ordered anywhere in the matrix, possibly causing large-size updates later on. To alleviate this problem, the proposed implementation constraints recently observed landmarks to immediately precede the last pose. Figures 7.1 and 7.2 show that the used ordering restrictions barely affect the fill-in. Furthermore, due to the inherent block structure, and in order to facilitate further incremental updates, the ordering is done by blocks. Figure 7.2 shows that applying ordering by blocks instead of elementwise has very small influence in the fill-in of the R factor. The small but persistent difference between the elementwise and blockwise orderings is caused mostly by the fact that the diagonal blocks in R are half empty, but still have to be stored as full blocks.

7.4.3 Fast Update Factorization

In the increment formula (7.8), a need arises to factorize a sparse block matrix. Note that this is slightly different from the batch solving where the aim was to solve a linear system. On the other hand, here we are interested in the factorization itself. In the proposed implementation, the Cholesky factorization is calculated using CSpase [41] or Cholmod [42] and then *converted back* to a sparse block matrix. This factorization is performed practically at every step and its speed affects the speed of the incremental solver. Fortunately, (7.8) is usually rather small and dense.

Algorithm 7.1: Incremental [SLAM](#) algorithm using the algebraic R updates.

```

1: function INCREMENTALR( $\theta, \mathbf{r}, \Sigma, \mathbf{R}, \mathbf{d}, \Lambda, \eta, \text{new}_{\text{LP}}, \text{max}_{\text{iters}}, \text{tol}$ )
2:    $(\Omega, \omega) \leftarrow \text{COMPUTEOMEGA}((\theta_{i_k}, \theta_{j_k}), r_k, \Sigma_k)$ 
3:   if  $\text{new}_{\text{LP}}$  then
4:      $(\hat{\Lambda}, \hat{\eta}) = \text{LINEARSYSTEM}(\theta, \mathbf{r})$ 
5:   else
6:      $(\hat{\Lambda}, \hat{\eta}) = \text{UPDATELINEARSYSTEM}(\Lambda, \eta, \Omega, \omega)$ 
7:   end if
8:    $\text{loopSize} = \text{SIZE}(\text{BLOCKCOLS}(\Omega))$ 
9:   if  $\text{new}_{\text{LP}}$  or  $\text{loopSize} > \text{bigLoopThresh}$  then
10:     $\hat{\mathbf{R}} = \text{chol}(\hat{\Lambda})$   $\triangleright$  Calculate variable ordering using constrained AMD.
11:     $\hat{\mathbf{d}} = \hat{\mathbf{R}} \setminus \hat{\eta}$ 
12:  else
13:    if  $\text{loopSize} < \text{smallLoopThresh}$  then
14:       $\hat{\mathbf{R}} = (\mathbf{R}_{11}, \mathbf{R}_{12}; 0, \text{chol}(\Omega + \mathbf{R}_{22}^{\top} \mathbf{R}_{22}))$ 
15:    else
16:       $\hat{\mathbf{R}} = (\mathbf{R}_{11}, \mathbf{R}_{12}; 0, \text{chol}(\hat{\Lambda}_{22} - \mathbf{R}_{12}^{\top} \mathbf{R}_{12}))$ 
17:    end if
18:     $\hat{\mathbf{d}} = (\mathbf{d}_1; \hat{\mathbf{R}}_{22}^{\top} \setminus (\hat{\eta}_2 - \mathbf{R}_{12}^{\top} \mathbf{d}_1))$ 
19:  end if
20:   $\text{new}_{\text{LP}} = \text{false}$   $\triangleright$  Both  $\hat{\Lambda}$  and  $\hat{\mathbf{R}}$  now contain the current linearization point.
21:  if  $\text{max}_{\text{iters}} \leq 0$  or  $\neg \text{hadLoop}$  then
22:    return
23:  end if
24:   $\text{GAUSSNEWTON}(\theta, \mathbf{r}, \Sigma, \hat{\mathbf{R}}, \hat{\mathbf{d}}, \hat{\Lambda}, \hat{\eta}, \text{new}_{\text{LP}}, \text{max}_{\text{iters}}, \text{tol})$ 
25: end function

```

Applying dense Cholesky is faster than sparse Cholesky, up to a certain limit where the dense implementation gets beaten by the fact that it operates mostly on zeroes when \mathbf{R} is very sparse. Therefore, dense Cholesky is applied for matrices up to 5×5 blocks which occur relatively frequently in (7.8). This Cholesky is further optimized by anticipating the possible combinations of the sizes of \mathbf{R}_{22} from the knowledge of the dimension of the variables. E.g. in 3D [SLAM](#), the variables have 6 [DOF](#) and therefore the possible matrices can be 6×6 , 12×12 and so on.

7.4.4 Incremental Algorithm

The proposed approach is described by pseudocode in [Algorithm 7.1](#). It can be understood as having three distinct parts. The first part (lines 3 to 7) is keeping

Algorithm 7.2: Gauss-Newton algorithm using the R factorization.

```

1: function GAUSSNEWTON( $\theta$ ,  $\mathbf{r}$ ,  $\Sigma$ ,  $\hat{\mathbf{R}}$ ,  $\hat{\mathbf{d}}$ ,  $\hat{\Lambda}$ ,  $\hat{\boldsymbol{\eta}}$ , newLP, maxiters, tol)
Require: This function assumes that newLP is initially false ( $\hat{\Lambda}$  and  $\hat{\mathbf{R}}$  are current).
2:   for it = 1 to maxiters do
3:     if it > 1 then
4:        $(\hat{\Lambda}, \hat{\boldsymbol{\eta}}) = \text{LINEARSYSTEM}(\theta, \mathbf{r})$ 
5:        $\hat{\mathbf{R}} = \text{chol}(\hat{\Lambda})$ 
6:        $\hat{\mathbf{d}} = \hat{\mathbf{R}} \setminus \hat{\boldsymbol{\eta}}$ 
7:       newLP = false
8:     end if
9:      $\boldsymbol{\delta} = \hat{\mathbf{R}} \setminus \hat{\mathbf{d}}$ 
10:    if  $\|\boldsymbol{\delta}\| \geq \text{tol}$  then
11:       $\theta = \theta \oplus \boldsymbol{\delta}$  ▷ The linearization point changes.
12:      newLP = true
13:    else
14:      break
15:    end if
16:  end for
17: end function

```

the Λ matrix up to date. This can be done incrementally by adding Ω , unless the linearization point changed. The change in the linearization point is stored in the `have Λ` flag.

The second part of the algorithm updates the R factor (lines 9 to 19). The algorithm employs a simple heuristic to decide which update method is the fastest. In case of large updates, invalidating a substantial portion of R, or if the linearization point has changed, $\hat{\mathbf{R}}$ is recalculated from $\hat{\Lambda}$. This step involves calculating a suitable variable ordering using the constrained AMD algorithm. On the other hand, if R was up to date, before the new observations were introduced into the system, and the size of the update is relatively small, it is faster to update $\hat{\mathbf{R}}$ using either (7.8), which is faster for smaller updates, or using (7.7). The r.h.s. vector $\hat{\mathbf{d}}$ is updated in a similar manner. Please, note that while the thresholds used in this part of the algorithm affect the speed of the computation, they do not affect the precision of the results in any way.

The final part of the algorithm is basically a simple Gauss-Newton nonlinear solver, listed separately in Algorithm 7.2. An interesting point to note is that the nonlinear solver only needs to run if the residual grew after the last update. This is due to two assumptions; one is that the allowed number of iterations max_{iters} is always sufficiently large to reach the local minima, and the other is that good initial

Table 7.1: Evaluation of incremental solving times (solve at each step) of the NLS optimizers on multiple datasets, in seconds (the best times of solutions which solve at each step are in bold). Note that Inc-R is using the algebraic Cholesky updates.

Dataset	Manhattan	10K	CityTrees10k	Intel	Killian C.	Victoria P.
SPA	23.8834	515.2880	N/A	1.4763	5.6260	N/A
g2o	94.9096	2134.3000	659.1590	5.0513	20.8899	293.1010
iSAM	64.5844	1768.8400	434.7500	4.4647	19.7519	209.1740
iSAM b ₁₀	9.9222	334.3650	60.2726	0.9442	3.6273	29.5268
iSAM b ₁₀₀	4.7142	289.7870	25.2429	1.3648	4.2522	12.6860
allBatch- Λ	10.0038	329.1840	22.7070	0.8424	2.1485	28.0194
Inc-R	5.0274	183.3850	25.5549	0.7032	2.5719	16.0173
Inc-R b ₁₀	5.0275	166.7970	25.3064	0.6861	2.4637	14.6821

priors are calculated. Without a loop closure, the norm of δ would be close to zero and the system would not be updated anyway. This information is introduced to the Algorithm 7.1 at line 21 (if such information is not available, it is safe to assume that there always is a loop closure, at a cost of an extra backsubstitution). The first iteration uses the updated \hat{R} factor, and the subsequent iterations rebuild both $\hat{\Lambda}$ and \hat{R} so that \hat{R} would be available for the next steps in case the algorithm finishes by reaching the solution.

7.5 EXPERIMENTAL RESULTS

In order to evaluate the proposed incremental algorithm and its implementation this section compares timing with similar state of the art implementations such as iSAM [95], g2o [106], and SPA [104] (a 2D SLAM variant of sSBA). These implementations are easy to use on standard datasets. iSAM2 [97, 98], on the other hand, is an incremental algorithm based on GTSAM library, and, at the time of running the benchmarks, the source code for iSAM2 was not available among the examples of the GTSAM library. The reported results from iSAM2 papers [97, 98] cannot be used for comparisons since they were measured on a radically different platform.

The evaluation was performed on three standard simulated datasets, *Manhattan*, [137], *10k* and *CityTrees10k*, [94] and on three real datasets, *Intel*, [85], *Killian Court*, [21] and *Victoria Park* [133] dataset. The solution for each dataset is shown in Figures 6.1 and 6.2.

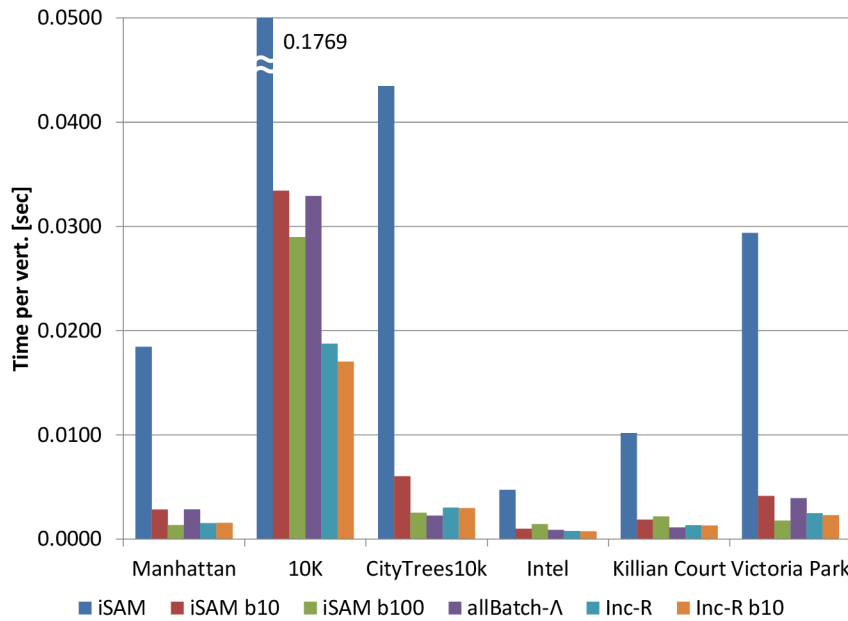


Figure 7.3: Time comparison of multiple NLS optimizers.

Again, the same machine as in the previous chapter was used for the tests, an Intel Core i5 CPU 661 with 8 GB of RAM running at 3.33 GHz. 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 datasets.

Table 7.1 and Figure 7.3 show the execution times of different implementations evaluated on the above mentioned datasets. The b10 and b100 flags represent the frequency of batch computations – once each 10 and once each 100 steps, respectively. For the results without those flags, the nonlinear system was solved at every step in order to obtain the current estimation or only when needed in the case of the proposed Incremental-R algorithm. Unlike g20 and SPA, iSAM and our implementation provide both the factorization and an error-minimizing solution at every step, even when the batch solver runs only each 10 or each 100 steps. This is an important characteristic for online applications. Therefore, and in order to make the spread of the plotted values lower, Figure 7.3 shows timing results only for iSAM and for the proposed implementation.

All the times below the double horizontal line in Table 7.1 are obtained using the proposed implementation. The execution time of Algorithm 7.1 is denoted Inc-R. The Inc-R b10 is obtained by forcing batch every 10, but observe that this is not the natural way to execute our algorithm and has been introduced only for comparison purposes. allBatch- Λ is an implementation of the algorithm introduced in Chapter 6 – it keeps and updates only the Λ matrix and performs matrix factorization every time a new linearization point needs to be calculated. From the point

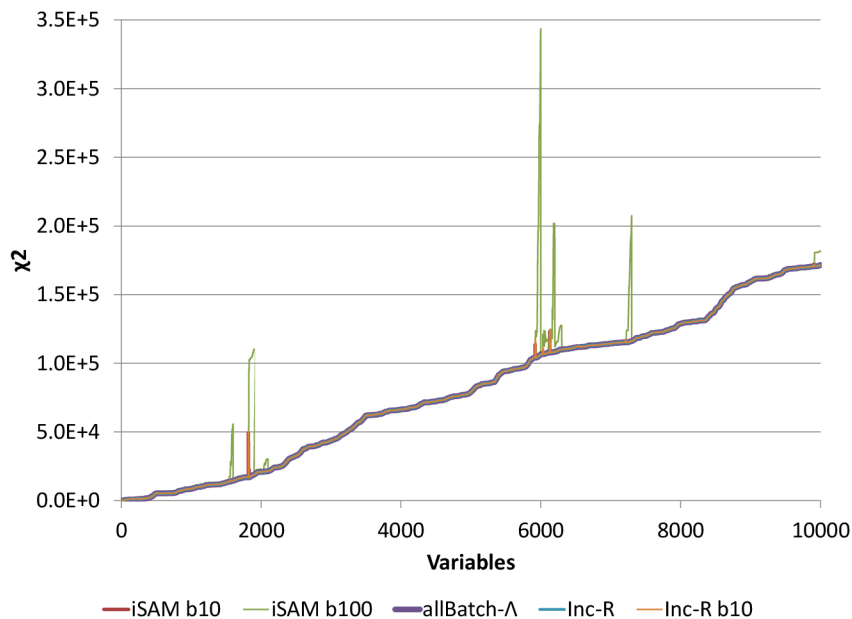


Figure 7.4: Comparison of the χ^2 errors, on the *10k* dataset.

of view of estimation quality, recalculating the system every time the linearization point changes, is the best the nonlinear solver can do but it can sometimes become computationally expensive. Even though, our optimized implementation performs very well also in the allBatch- Λ case.

Figure 7.4 compares the quality of the estimations measured by the sum of squared errors, the χ^2 errors. The test was performed for the *10k* dataset. Observe that our new algorithm, Inc-R (in orange in Figure 7.4), nicely follows the allBatch- Λ (in violet in Figure 7.4). Spikes appear when performing periodic batch solve in iSAM b100, iSAM b10 and Inc-R b10 due to the fact that the error increases between the batch steps and drops afterwards.

As an overall remark, the Inc-R has, in general, the best performance (which is only rivaled by allBatch- Λ from the previous chapter) and provides very accurate results every step. Compared to allBatch- Λ , it provides not only the solution but also the factorization at every step. That amounts to doing slightly more work, but allows doing one more Gauss-Newton step towards the solution, at virtually no cost. It also becomes important if the *covariances* of the solution need to be recovered as well. Therefore, it is the most suitable implementation for online applications which require efficient nonlinear least squares solving.

7.6 IMPROVED ALGORITHM USING BLOCK CHOLESKY FACTORIZATION

The incremental algorithm described so far made use of block matrix operations, *except* for the block Cholesky factorization. It needed to convert the Λ matrix to elementwise sparse one, factorize it using CSpase and then convert the factor back

to blockwise representation. Although competitive, the incremental implementation is really taking the toll by performing this conversion at each step. Another disadvantage is its inability to reorder the variables in the factorization, after e.g. a loop closure occurs. It only relies on reordering when linearization point changes take place (they usually happen at loop closures) and on cleverly constraining the ordering in order to be able to efficiently update the factorization while going in an open loop.

While the implementation described above was comparable with the others of its time, Kaess et al. later introduced the Bayes tree data structure [98], which provides insights on the connection between graphical model inference and sparse matrix factorization. This offered the possibility of eliminating the periodic batch steps by allowing incremental variable re-ordering to reduce the fill-in and implementing fluid relinearization to guarantee good linearization points [97]. In the remaining part of this chapter, an improved incremental algorithm which takes advantage of the sparse block Cholesky factorization from Section 5.2.4 is described and compared yet again to the state of the art solvers.

The work introduced in the paragraphs below combines the efficiency of operating directly on the matrix factorization with the insights gained from the Bayes tree data structure to produce highly efficient incremental solutions. The incremental solution proposed here is changing the linearization point every time if the error increases. This guarantees high quality estimates. Furthermore, it is based on a *resumed*³ Cholesky factorization which recalculates only the parts affected by the new updates, together with an incremental reordering scheme which maintains the factorization sparse without the need for periodic batch steps.

This form of incrementally updating the Cholesky factor is very similar to the incremental updates proposed in [95], where the authors use Q-less QR factorization to incrementally factorize R. In its form, this factorization is de-facto resumed: the factor R is calculated by transforming rows of A by Givens rotations into R. After new observations are made, these are added as new rows to yield \hat{A} . The factorization is then resumed at the first of these new rows, adding them to \hat{R} . Similar row-oriented methods are used for out-of-core QR factorizations of large systems.

However, this type of QR factorization does not make it possible to reorder the variables: A is ordered using column ordering. Therefore, reordering the columns

³ In the context of iterative numerical methods and subspace methods, the word *restarted* is sometimes used, meaning that the algorithm can stop iterating at some point and then be restarted later, possibly in different conditions. Our use of the word *resumed* refers to a direct method involving Cholesky factorization. Our implementation of Cholesky is left-looking and produces one column of the factor at a time. If the right part of the original matrix changes later, the factorization can be started in the middle (resumed), at the first column that will change to recalculate only the corresponding right portion of the factor while keeping the left part intact and saving computation.

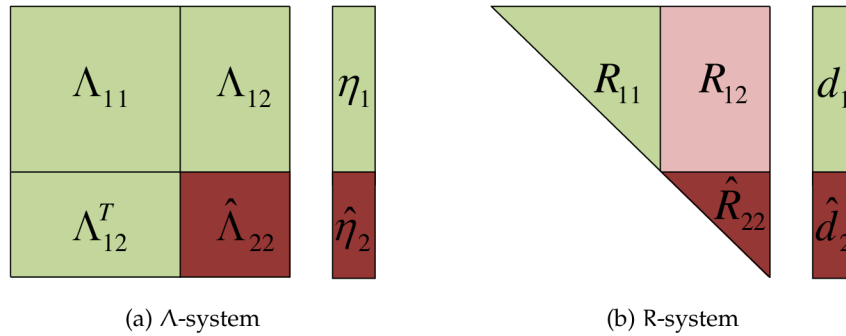


Figure 7.5: Incremental updates on Λ -system and R-system. The sections are color-coded according to the way they are affected by the update. Green – not affected, red – affected, pink – affected if the variables are also reordered.

potentially affects all the rows, making the tracking of changes in the factorization in order to reuse the unaffected parts infeasible. The recently introduced data structure, the Bayes tree [98], offers the possibility to develop incremental algorithms where variable reordering can be performed fluidly. Inspired by these recent advances, the resumed Cholesky factorization is an elegant and highly efficient solution which combines the efficiency of block matrix implementation and considers the insights gained using the Bayes tree data structure.

Our implementation maximally exploits the sparse block structure of the problem. On one hand, the block matrix manipulation is highly optimized, facilitating structural and numerical matrix changes while also performing arithmetic operations efficiently. On the other hand, the block structure is maintained in all the operations including the matrix factorization, eliminating the cost of converting between sparse elementwise and sparse blockwise. Our block Cholesky factorization implementation proves to be significantly faster than the existing state of the art elementwise implementations.

7.7 INCREMENTAL UPDATES OF THE FACTOR USING RESUMED CHOLESKY

Similarly as in Section 7.3, the task at hand is updating the Cholesky factor after new measurements have been added to the system (in case the added measurements involve new variables, the Λ and R matrices are first augmented with zero block rows and zero block columns, with their number and size corresponding to the number and DOF of each new variable). It is still possible to use equations (7.7) and (7.8) and a subsequent factorization to achieve that. It was already demonstrated that these only yield changes in Λ_{22} but it was not shown how these affect the factor. Figure 7.5 shows these changes in both Λ and also R; the parts corresponding to entries not affected by the update are marked green and the affected

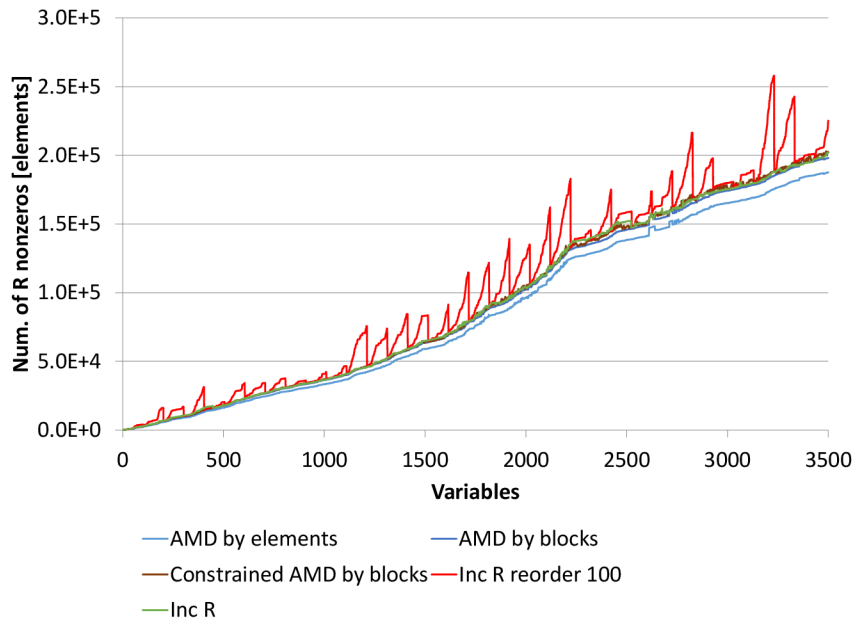


Figure 7.6: Evaluation of several ordering heuristics in terms of nonzero elements compared to the actual number of non-zero elements in the Incremental R algorithm. This is on the *Manhattan* dataset.

parts are marked red. The associated cost depends on the size of the update (the number of columns in Ω or equally in $\hat{\Lambda}_{22}$) but also – and often more importantly – on the sparsity of the resulting factorization \hat{R}_{22} .

In [SLAM](#), the size of the update is typically small since the new observations tend to link variables recently added to the system, but in general, it can become very large if the new observations link variables far apart (such as in loop closures). It is impossible to guess which variables are going to be linked in the future and thus the size of the update cannot be directly minimized. Ordering the recent variables last as suggested e.g. in [\[95\]](#) helps, but it is not a universal remedy.

On the other hand, there are efficient heuristics for variable reordering which minimize fill-in and increase sparsity in the subsequent factorization, e.g. Approximate Minimum Degree (AMD) [\[6\]](#). It is therefore possible to reorder the variables involved in the update, so as to minimize the fill-in caused by observations that link variables far apart. Once the variables involved in the update were reordered, R_{21} also needs to be recalculated, in addition to R_{22} . This is illustrated in [Figure 7.5](#) using pink color. The following subsection describes how this reordering can be calculated incrementally.

7.7.1 Incremental Ordering

[Section 5.2.4](#) introduced some of the commonly used ordering heuristics which are directly applicable to batch factorization. Additionally, [Section 7.4.2](#) showed how

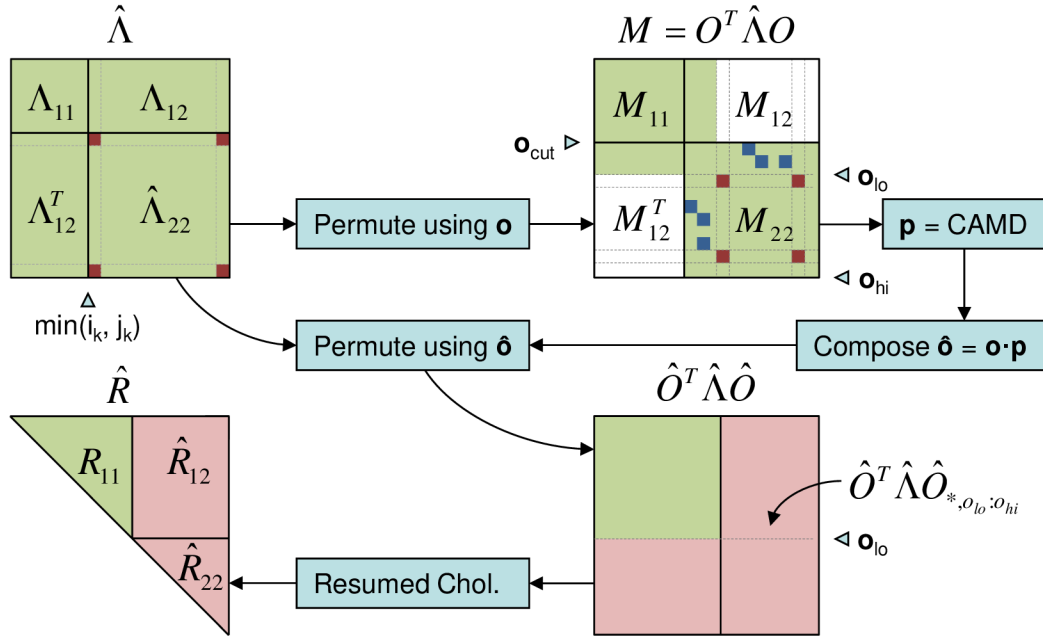


Figure 7.7: Dataflow diagram of incremental block Cholesky factorization. Green parts of the matrices do not change, red parts represent the update and pink represents the parts that will change. White parts are zero. The explanation is simplified to updates involving only two variables. Note that the green parts in $M = O^T \hat{\Lambda} O$ and in $\hat{O}^T \hat{\Lambda} \hat{O}$ are unchanged with respect to the previous step rather than with respect to the unpermuted $\hat{\Lambda}$.

constrained ordering can be applied to improve the locality of the new measurements added to the system. In order to efficiently maintain incremental factorization, incremental variable ordering is considered. Note that so far, the sparsity of the updates in Λ were considered under the *natural* ordering (the order in which the variables are observed and introduced into the system).

In this section, a permutation matrix O is introduced, which contains the fill-reducing ordering. In the implementation, it is represented in its vectorial form by the variable number reassignment vector \mathbf{o} . This ordering is maintained incrementally, along with Λ and R . So far, the fill-reducing ordering was only implied. For the remainder of this section, Λ and $O^T \Lambda O$ are written explicitly, with $R \triangleq \text{chol}(O^T \Lambda O)$ and $\hat{R} \triangleq \text{chol}(\hat{O}^T \hat{\Lambda} \hat{O})$.

The proposed incremental ordering solution is to only calculate the new ordering for parts of R which are being affected by the update. In order to be able to calculate the new ordering \hat{O} incrementally, the updated $\hat{\Lambda}$ matrix is first permuted with the ordering from the previous step, leading to $M \triangleq O^T \hat{\Lambda} O$ (see Figure 7.7). The ordering increment P is then calculated on this matrix, and composed with the old ordering to yield $\hat{O} = O \cdot P$ (here the multiplication denotes composition).

To delimit the area in $M = O^T \hat{\Lambda} O$ affected by the update, two indices are introduced. The first one, \mathbf{o}_{lo} is given by the minimum variable index after the ordering

Algorithm 7.3: Incremental Block Cholesky Factorization.

```

1: function UPDATER( $i_k, j_k, \Omega, \omega, R, \hat{\Lambda}, \hat{\eta}, \mathbf{o}, \text{new}_{LP}$ )
2:   if  $\text{new}_{LP}$  or  $\text{COLS}(\Omega) = \text{COLS}(\hat{\Lambda})$  then
3:      $\hat{\mathbf{o}} = \text{CAMD}(\hat{\Lambda}, \text{last variable constraints})$ 
4:      $\hat{R} = \text{chol}(\text{PERMUTE}(\hat{\Lambda}, \hat{\mathbf{o}}))$   $\triangleright \text{PERMUTE}(\hat{\Lambda}, \hat{\mathbf{o}})$  produces  $\hat{O}^\top \hat{\Lambda} \hat{O}$ .
5:      $\hat{\mathbf{d}} = \hat{R} \setminus \hat{\eta}$   $\triangleright$  Makes appropriate use of the ordering  $\hat{\mathbf{o}}$ .
6:   else
7:      $o_{lo} = \min(\mathbf{o}_{i_k}, \mathbf{o}_{j_k})$ 
8:      $o_{hi} = \text{SIZE}(\text{BLOCKCOLS}(\hat{\Lambda}))$ 
9:      $M = \text{PERMUTE}(\hat{\Lambda}, \mathbf{o})$   $\triangleright \text{PERMUTE}(\hat{\Lambda}, \mathbf{o})$  produces  $O^\top \hat{\Lambda} O$ .
10:     $o_{cut} = \min_{i=o_{lo}}^{o_{hi}} (\text{WAVEFRONT}(M)_i)$ 
11:     $M_{22} = M_{o_{cut}:o_{hi}, o_{cut}:o_{hi}}$ 
12:     $\mathbf{p} = \text{CAMD}(M_{22}, \text{constrain } o_{lo} - o_{cut} \text{ first elements})_{o_{lo}-o_{cut}:\text{end}}$ 
13:     $\hat{\mathbf{o}} = [\mathbf{o}_{0:o_{lo}}, \text{COMPOSE}(\mathbf{o}_{o_{lo}:o_{hi}}, \mathbf{p})]$ 
14:    if  $\mathbf{p} \neq \text{identity}$  then
15:       $R_{*, o_{lo}:o_{hi}} = \text{PERMUTE}(\hat{\Lambda}, \hat{\mathbf{o}})_{*, o_{lo}:o_{hi}}$   $\triangleright$  Splice right half of  $\hat{O}^\top \hat{\Lambda} \hat{O}$  to  $R$ .
16:       $\hat{R} = \text{RESUMEDCHOL}(R, o_{lo})$   $\triangleright$  Cholesky of  $R$ , starting at column  $o_{lo}$ .
17:       $\mathbf{d}_{o_{lo}:o_{hi}} = \hat{\eta}_{o_{lo}:o_{hi}}$   $\triangleright$  Splice the bottom part of  $\hat{\eta}$  to  $\mathbf{d}$ .
18:       $\hat{\mathbf{d}} = \text{RESUMEDUSOLVE}(\hat{R}_{*, o_{lo}:o_{hi}}^\top, \mathbf{d}, o_{lo})$   $\triangleright$  Resumed backsubstitution.
19:    else
20:       $\hat{R} = [R_{11}, R_{12}; 0, \text{chol}(\Omega + R_{22}^\top R_{22})]$ 
21:       $\hat{\mathbf{d}} = [\mathbf{d}_1, \hat{R}_{22}^\top \setminus (\hat{\eta}_2 - \hat{R}_{21}^\top \mathbf{d}_1)]$   $\triangleright$  Makes appropriate use of  $\hat{\mathbf{o}}$ .
22:    end if
23:  end if
24:  return  $(\hat{R}, \hat{\mathbf{d}}, \hat{\mathbf{o}})$ 
25: end function

```

(line 7 of the Algorithm 7.3). The second one, o_{hi} , is simply the size of the matrix. Let $M_{o_{lo}:o_{hi}, o_{lo}:o_{hi}}$ be the lower right submatrix of M delimited by those indices. In case the ordering is identity, this submatrix matches $\hat{\Lambda}_{22}$ – but generally $O \neq I$ and so those are two different matrices of different size.

Calculating the ordering update as AMD on $M_{o_{lo}:o_{hi}, o_{lo}:o_{hi}}$ is not sufficient and also leads to massive fill-in. This is caused by the AMD algorithm not having any information about the nonzero entries in $M_{1:o_{lo}-1, o_{lo}:o_{hi}} = M_{o_{lo}:o_{hi}, 1:o_{lo}-1}^\top$, which are also affected by this ordering (depicted by the blue blocks in Figure 7.7).

A better ordering can be calculated as AMD of full M with constraints applied to ensure that the order of the variables unaffected by the update stays the same. This is however computationally expensive, since the update is typically much smaller than M and thus a relatively large number of ordering constraints is needed.

Fortunately, it is not necessary to calculate the ordering using the entire M . It is possible to use a slightly expanded $M_{22} \triangleq M_{o_{\text{cut}}:o_{\text{hi}}, o_{\text{cut}}:o_{\text{hi}}}$ (see [Figure 7.7](#)) that satisfies the conditions of being square and not having any nonzero elements above or left from it (so that $M_{1:o_{\text{cut}}-1, o_{\text{cut}}:o_{\text{hi}}} = M_{o_{\text{cut}}:o_{\text{hi}}, 1:o_{\text{cut}}-1}^\top$ which correspond to the right and bottom portions of M_{12} and M_{12}^\top , respectively, are null). The ordering calculated on this submatrix is then combined with the original ordering (lines [11](#) through [13](#) in [Algorithm 7.3](#)), yielding a similar result as constrained ordering on full M in much smaller time. The minimal size of the expanded M_{22} can be calculated in linear time $O(o_{\text{hi}} - o_{\text{cut}})$. First, a matrix *wavefront* is calculated. This is a vector containing the block row indices of the first nonzero block per each block column of M . Only a part of this vector is used, the one between o_{lo} and o_{hi} , and its minimum gives the index of the highest nonzero element, o_{cut} . This is done in [Algorithm 7.3](#) on line [10](#), and in the [Figure 7.7](#) top right it is depicted as the line, keeping the blue nonzero blocks out of M_{12} . Extending M_{22} makes [AMD](#) aware of all the nonzero elements that would affect the fill-in, leading to a better ordering.

Once the new ordering is calculated, factorization can be performed. In case that the ordering is identity, it is possible to only update R_{22} and d_2 using [\(7.8\)](#) and [\(7.10\)](#). Otherwise, the *resumed Cholesky* algorithm is employed (line [16](#) in [Algorithm 7.3](#)). The column Cholesky (a sparse case of the one in [Algorithm 5.4](#)) is capable of calculating one column of the factor at a time, while only reading the values to the left from it. This algorithm can be modified to be able to “resume” the factorization in the right part of R while only using the corresponding part of $(O^\top \hat{\Lambda} O)_{*,2}$ and R_{11} as inputs. The advantage of this algorithm is overall simplicity of the incremental updates to the factor, while also saving substantial time by avoiding the recalculation of \hat{R}_{11} , compared to the batch approach. Another advantage is higher numerical stability, compared to rank up- and downdate where near semidefinite matrices can occur and numerical errors can accumulate over time.

Note that at line [17](#) in [Algorithm 7.3](#), resumed backsubstitution is employed. It could easily be replaced by full backsubstitution to achieve the same result, but similarly to the resumed Cholesky factorization, resuming solving can save some computation as well since some of the entries do not change. This is vaguely similar to the strategy of [iSAM2](#) [[97](#), [98](#)] which thresholds the nodes of the Bayes tree when solving, so that branches which only contribute small change in the solution of the linearized system (i.e. $\|\hat{\delta} - \delta\|$ rather than δ) could be skipped.

7.7.2 Incremental SLAM Algorithm

The improved approach to incremental [SLAM](#) is described by the pseudocode in [Algorithm 7.4](#), which can be seen as having three distinct parts. The first part is

Algorithm 7.4: Improved incremental SLAM algorithm using resumed Cholesky.

```

1: function FASTINCREMENTALR( $\theta, \mathbf{r}, \Sigma, \mathbf{R}, \mathbf{d}, \Lambda, \boldsymbol{\eta}, \mathbf{o}, \text{new}_{\text{LP}}, \text{max}_{\text{iters}}, \text{tol}$ )
2:    $(\Omega, \boldsymbol{\omega}) \leftarrow \text{COMPUTEOMEGA}((\theta_{i_k}, \theta_{j_k}), \mathbf{r}_k, \Sigma_k)$ 
3:   if  $\text{new}_{\text{LP}}$  then
4:      $(\hat{\Lambda}, \hat{\boldsymbol{\eta}}) = \text{LINEARSYSTEM}(\boldsymbol{\theta}, \mathbf{r})$ 
5:   else
6:      $(\hat{\Lambda}, \hat{\boldsymbol{\eta}}) = \text{UPDATELINEARSYSTEM}(\Lambda, \boldsymbol{\eta}, \Omega, \boldsymbol{\omega})$ 
7:   end if
8:    $(\hat{\mathbf{R}}, \hat{\mathbf{d}}, \hat{\boldsymbol{\delta}}) = \text{UPDATER}(i_k, j_k, \Omega, \boldsymbol{\omega}, \mathbf{R}, \hat{\Lambda}, \hat{\boldsymbol{\eta}}, \mathbf{o}, \text{new}_{\text{LP}})$ 
9:    $\text{new}_{\text{LP}} = \text{false}$ 
10:  if  $\text{max}_{\text{iters}} \leq 0$  or  $\neg \text{hadLoop}$  then
11:    return
12:  end if
13:   $\text{GAUSSNEWTON}(\boldsymbol{\theta}, \mathbf{r}, \Sigma, \hat{\mathbf{R}}, \hat{\mathbf{d}}, \hat{\Lambda}, \hat{\boldsymbol{\eta}}, \text{new}_{\text{LP}}, \text{max}_{\text{iters}}, \text{tol})$  ▷ Algorithm 7.2.
14: end function

```

keeping the Λ matrix up to date. This can be done incrementally by adding Ω , unless the linearization point changed. The change in the linearization point is stored in the new_{LP} flag. The second part of the algorithm updates the \mathbf{R} factor along with the associated ordering, as described in the previous section. The third part of the algorithm is again a simple Gauss-Newton nonlinear solver. An important point to note is that the nonlinear solver only needs to run if the residual grew after the last update. This is due to two assumptions; one is that the allowed number of iterations $\text{max}_{\text{iters}}$ is always sufficiently large to reach the local minima, and the other is that good initial priors are calculated. Without a loop closure, the norm of $\boldsymbol{\delta}$ would be close to zero and the system would not be updated. The first iteration uses the incrementally updated \mathbf{R} factor, and the subsequent iterations calculate \mathbf{R} as $\text{batch chol}(\Lambda)$ since the linearization point has changed.

7.8 EXPERIMENTAL EVALUATION

This section evaluates both, the implementation of the incremental algorithm and of the incremental block Cholesky factorization by comparing timing and the quality of the result with similar state of the art implementations. The evaluation was performed on five standard simulated datasets, *Manhattan* [137], *10k*, *City10k* [94], *CityTrees10k* [94] and *Sphere* [106] and four real datasets, *Intel* [85], *Killian Court* [21], *Victoria Park* and *Parking Garage* [106]. Figure 7.8 shows the final solutions for all the tested datasets. All the tests were performed on an Intel Core i5 CPU 661 with 8 GB of RAM and running at 3.33 GHz, much like the benchmarks in previous chap-

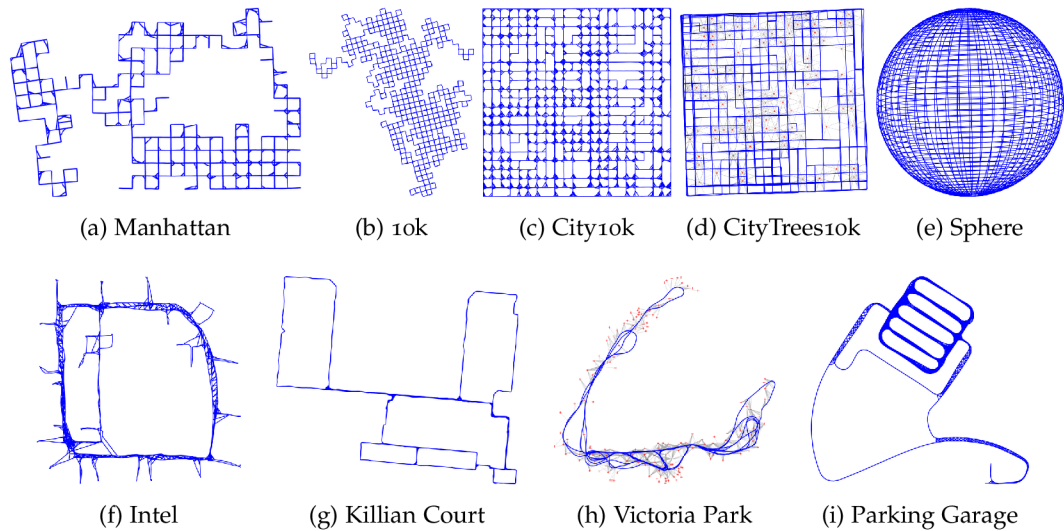


Figure 7.8: SLAM datasets used in the evaluations. The top row are synthetic datasets, real datasets are in the bottom row. *Sphere* and *Parking Garage* datasets are 3D pose graphs.

ters. 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.

7.8.1 Tested Implementations

We compared the proposed incremental algorithm and its implementation with state of the art implementations such as g2o [106], iSAM [95] and the gtsam implementation of the iSAM2 algorithm [97, 98]. For SPA the SVN revision 39478 of ROS (<http://www.ros.org/>) was used; for g2o, we tested the version 91A858D available at <https://github.com/RainerKuemmerle/g2o>¹. For iSAM, version 1.6 from <https://svn.csail.mit.edu/isam> was used and for GTSAM, version 2.3 from <https://collab.cc.gatech.edu/borg/gtsam> was used.

SPA and g2o are both based on similar sparse block matrix scheme which is maintained until the matrix factorization is performed. At this point, they switch to CSC format to be able to use CSpase or Cholmod to perform the factorization. This is a time consuming process which is avoided in our approach. While SPA implementation is optimized for the specific 2D SLAM problem, g2o is general, allowing any type of SLAM and BA. Additionally, note that the incremental version of g2o is being used, which is much faster than the one in Tables 6.2 and 7.1.

¹ We thank the authors for providing the link and support

Table 7.2: Performance and accuracy tests on the simulated datasets. †Note that for 2D landmark datasets, iSAM2 and Inc-R use different parameterization and the χ^2 values are not comparable.

Dataset	Manhattan	10K	City10k	CityTrees10k	Sphere
SPA	24.161	518.339	309.562	N/A	N/A
g20	22.514	500.374	302.495	175.124	145.486
iSAM b ₁₀₀	4.829	279.926	77.572	22.926	36.220
iSAM ₂	4.932	91.738	60.978	32.687	31.274
allBatch- Λ -CS	8.603	287.702	202.839	19.531	216.487
allBatch- Λ -CM	10.725	236.276	181.139	24.478	71.487
allBatch- Λ -BC	7.209	242.209	188.849	17.566	78.375
Inc-R-BC	3.046	79.651	53.951	19.308	9.865
χ^2 iSAM ₂	6205.92	171600.00	31951.60	794.87	775.28
χ^2 Inc-R-BC	6119.83	171919.00	31931.40	12062.60 [†]	727.72

iSAM and iSAM₂ are based on completely different algorithms. The one used in iSAM is very similar to our algorithm but it requires periodic batch steps to reduce the fill-in. The algorithm used in iSAM₂ is based on the Bayes tree data structure and the factorization is done through elimination on factor graphs. One important characteristic is that it allows incremental reordering and fluid relinearization. In this direction, our algorithm allows similar incremental reordering but changes the entire linearization point when needed. In order to test the iSAM₂ we used the incremental test example provided by its authors and extended it to work with landmark-based and 3D SLAM datasets.

The proposed block Cholesky (BC) factorization is part of a new nonlinear least squares open-source library called SLAM ++, which is available for download at <http://sf.net/p/slam-plus-plus/>. The main characteristic of this new library is its ability to manipulate block matrices and to produce efficient incremental solutions. In this chapter we test the BC factorization on both, an algorithm that operates only on the information matrix Λ performing batch updates every step (denoted allBatch- Λ) and an incremental algorithm, which maintains the matrix factorization R up to date (denoted Inc-R). The latter corresponds to Algorithm 7.4.

The new library offers the possibility to switch between the “native” block Cholesky (BC) factorization and the Cholesky factorization form CSparse (CS) and Cholmod (CM). Those factorizations are compared on the allBatch- Λ algorithm which is relatively efficient even with elementwise factorization.

Table 7.3: Performance and accuracy tests on the real-world datasets. [†]Note that for 2D landmark datasets, iSAM2 and Inc-R use different parameterization and the χ^2 values are not comparable.

Dataset	Intel	Killian Court	Victoria Park	Parking Garage
SPA	1.486	5.669	N/A	N/A
g2o	1.298	5.019	81.194	20.372
iSAM b ₁₀₀	1.287	4.213	11.921	52.217
iSAM2	0.618	1.196	16.349	3.658
allBatch- Λ -CS	0.651	1.705	23.162	17.317
allBatch- Λ -CM	0.786	2.100	28.264	23.929
allBatch- Λ -BC	0.508	1.242	18.707	11.342
Inc-R-BC	0.353	1.045	11.202	3.410
χ^2 iSAM2	559.07	$8 \cdot 10^{-5}$	370.14	1.26
χ^2 Inc-R-BC	558.83	$5 \cdot 10^{-5}$	144.91 [†]	1.31

7.8.2 Performance and Accuracy

Tables 7.2 and 7.3 show the execution times and accuracy of the above described implementations evaluated on the simulated and real datasets in Figure 7.8, respectively. For every test, both building the system and computing the solution are part of the evaluation. The first one is necessary because changing the matrix numerically and structurally is different for each implementation and this makes significant difference in an incremental approach.

The proposed incremental algorithm is different from the one employed in SPA and g2o or in our allBatch- Λ solver, where the batch solving is done once every n new variables added to the system and no error reduction takes place in between. Therefore, the time comparison with these implementations is orientative. The comparison holds only for $n = 1$, where the solution is available at every step. iSAM, iSAM2 and Inc-R provide solution every step. The main difference is that iSAM requires the periodic batch solves, the default setting of $n = 100$ is used in the comparison. But keeping the same linearization point for too long deteriorates the estimation. This can be seen by plotting the the sum of squared errors (χ^2 in Figure 7.9). Spikes appear when performing periodic batch solve due to the fact that the error increases between the batch steps and drops afterwards. Observe that Inc-R (in red in Figure 7.9) and iSAM2 nicely follow the allBatch- Λ (in green in Figure 7.9), which represents the baseline, most accurate solution.

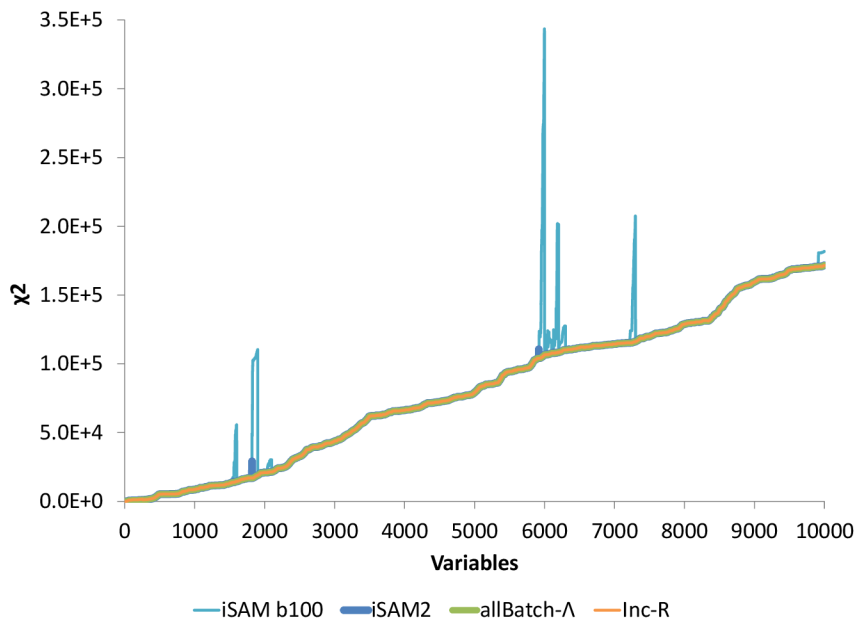


Figure 7.9: Quality of the estimations measured on the *10k* dataset.

The improved implementation reaches the best times for the best accuracy on all evaluated datasets and this is shown in bold in Tables 7.2 and 7.3. Except for the *CityTrees10k* dataset, the execution of the Inc-R outperforms all the implementations. This particular result is given by the dense structure of the problem. In this case, reordering every step is slightly more advantageous than incremental ordering. The closest time to Inc-R is reached by the iSAM2. The difference between iSAM2 and Inc-R is that iSAM2 changes only the affected blocks of the R factor and relinearizes only affected variables at each 10^{th} step, while Inc-R changes parts of the R factor and relinearizes all the affected variables when needed (iSAM2 was run with the default relinearization threshold 10). This leads to slightly worse accuracy of the estimation compared to Inc-R (see the last two rows of Tables 7.2 and 7.3) but makes iSAM2 run faster than if it was relinearizing at each step.

The proposed sparse block Cholesky factorization algorithm was tested on full system matrices of the same datasets used in the incremental algorithm evaluation. The results are shown in Figure 7.10a. The proposed block Cholesky implementation is always faster than the CSparse (v3.0.2) and is highly competitive with Cholmod (v2.1.2) which is only better on the *100k* and *Sphere* datasets where it takes advantage of large supernodes. Simplicial Cholmod is always slower. Also note that the speedup grows with the block size, for 6×6 blocks it is more than double. The quality of the factorization is also good, the worst norm of difference between block Cholesky and CSparse was $2.6016 \cdot 10^{-13}$ and occurred on the *City10k* dataset.

The speedups get slightly bigger in linear solving in Figure 7.10b. Here, back-substitution is performed along with fill-reducing ordering, Cholesky factorization

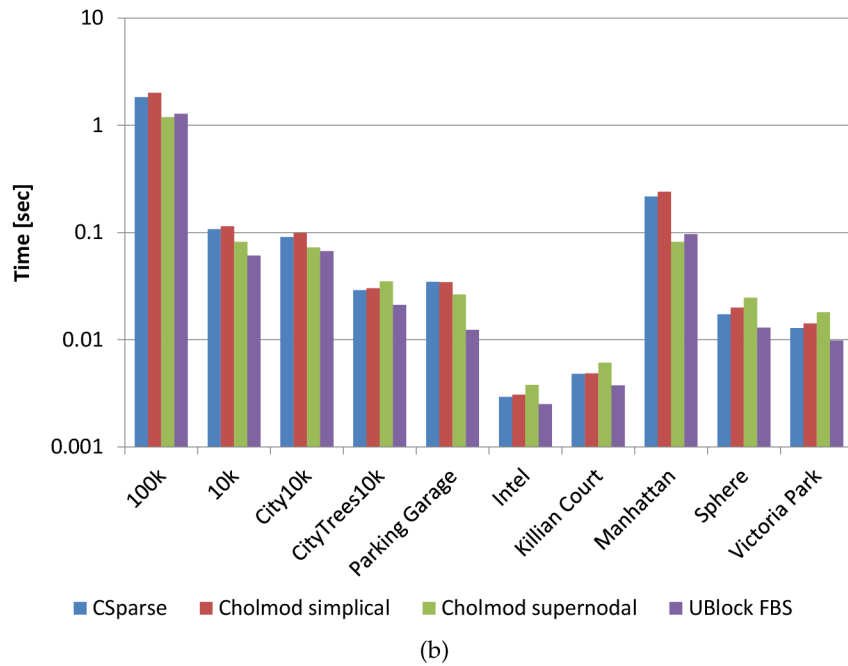
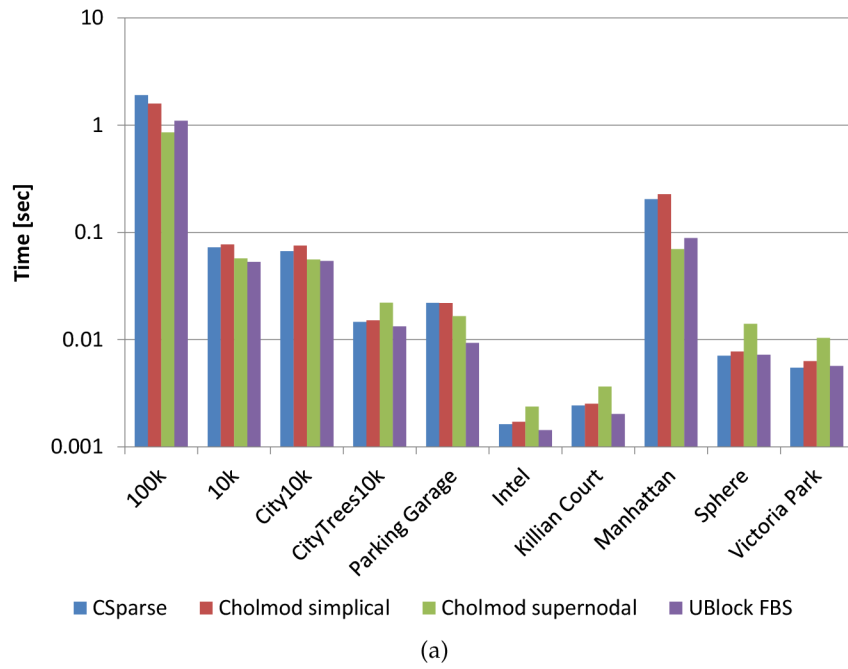


Figure 7.10: Cholesky factorization benchmark on the standard [SLAM](#) datasets, a) times of factorization only and b) times of linear solving.

(and block to sparse matrix conversion for CSparse and Cholmod). This benchmark is relevant because it demonstrates the real performance loss many state of the art [NLS](#) solvers pay by not using blockwise representation all the way through.

7.9 CHAPTER SUMMARY

A new incremental [NLS](#) algorithm with applications to robotics was proposed in this chapter. We targeted problems such as [SLAM](#), which have a particular block

structure, where the size of the blocks corresponds to the number of Degree of Freedom of the variables. This enabled several optimizations which made the proposed implementation faster than the state of the art implementations, while achieving very good precision. This was demonstrated through the comparison with the existing implementations on several standard datasets. While many of the algorithms from the iSAM family contain several thresholds and approximations. In contrast, our algorithm only retains a single threshold on $\|\delta\|$ which yields very precise solutions.

Recently, many efforts have been made to develop both, efficient incremental algorithms and implementations. This chapter complements the recent advances by introducing new incremental ordering scheme which allows to incrementally update the factorized form of the linearized system while maintaining a reduced fill-in. The incremental updates are done using a resumed block Cholesky factorization only on the parts affected by the new information. The block Cholesky factorization itself proved to be more efficient than the current implementations of elementwise Cholesky factorizations while the precision is equally high.

There are several areas for possible improvements. In the algebraic update method, the changed parts are treated as monolithic and there is little difference between situations where only a single constraint is added or where many constraints are added, as long as Ω is the same size. Computation can be saved by taking advantage of the sparsity of the updates and explicitly tracking the affected variables.

Similarly, the resumed Cholesky assumes that all of the right part of the factorization will change. But not all the columns of the factor may depend on the previous ones and thus not all need to actually be recalculated. Tracking the column dependences through the elimination tree could thus save computation in some cases when the respective ordering of the dependent columns does not change.

Finally, conventional rank updates could be implemented using our block scheme as well. This possibility was not explored as it would essentially lead to reimplementing of the iSAM algorithm.

SOLVING BUNDLE ADJUSTMENT PROBLEMS

While the efficient NLS solutions described earlier could readily be used to solve Bundle Adjustment (BA) problems, advantage can be taken of the structure of such problems. Applying Schur complement is one of the common optimizations, as already mentioned in Section 2.3. This chapter reviews the implementation of the Schur complement methods and their efficiency in solving BA – but also other problems, by using appropriate variable orderings.

In our context, the estimation problem is formulated as a Maximum Likelihood Estimation (MLE) of a set of variables $\theta = [\theta_1 \dots \theta_n]$ given a set of observations $z = [z_1 \dots z_m]$, much like already described in Section 2.1. Without the loss of generality, it is possible to order the variables in such a way that $\theta_1 \dots \theta_p$ are the p camera poses and $\theta_{p+1} \dots \theta_{n=p+1+l}$ are the l landmark positions and to assume that each constraint is between a pose variable and a landmark variable. Situations with additional types of variables (e.g. the intrinsic camera parameters) are possible. Situations with only a single type of variable (e.g. as in pose graph optimization) are also possible, although the ordering for Schur complement is more elaborate; one needs to compute the bipartite coloring if one exists or resort to maximum independent set if it does not.

By taking advantage of the structure of the problem, rather than solving the normal equation directly using a sparse factorization solver, it is possible to employ the Schur complement trick. In case the poses are ordered first, followed by all the landmarks, the normal equation (2.10) can be partitioned as:

$$\begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^\top & \Lambda_{22} \end{pmatrix} \cdot \begin{pmatrix} \delta_1 \\ \delta_2 \end{pmatrix} = \begin{pmatrix} \eta_1 \\ \eta_2 \end{pmatrix} \quad \text{or} \quad (8.1)$$

$$\begin{pmatrix} A & U \\ U^\top & D \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix},$$

where the D is supposed to be invertible and also block diagonal (since there are no observations that would directly relate two landmark variables and therefore no off-diagonal blocks are filled). See Figures 8.1b and 8.1e for examples of matrices from *Venice* [106] and *Guildford Cathedral*¹ datasets: the typical arrow shape shows

¹ can be obtained at <http://cvssp.org/impart/>

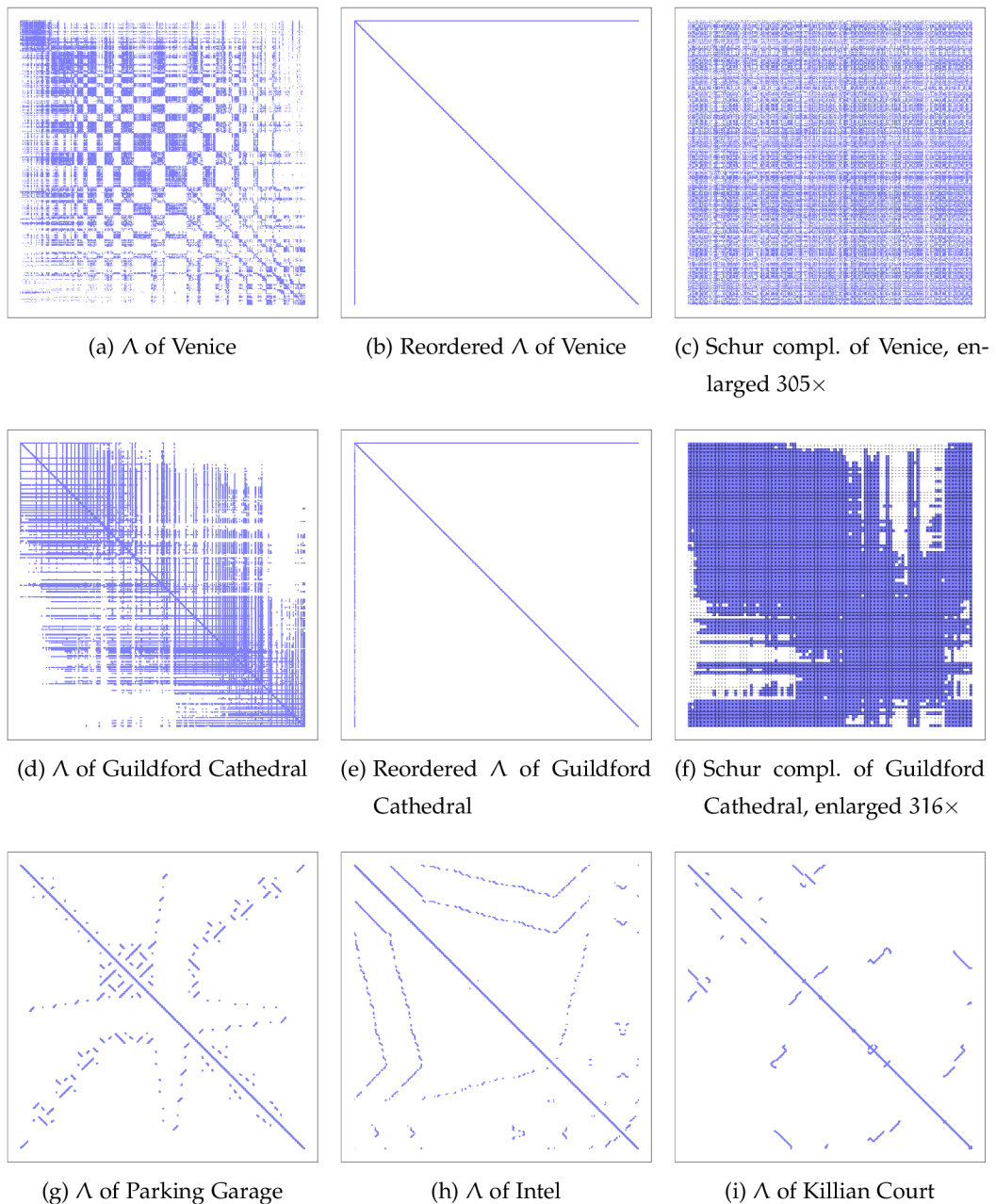


Figure 8.1: Sparsity patterns involved in common **BA** datasets (the first two rows) in contrast to **SLAM** (the bottom row) datasets. Note that each nonzero is inflated so as to be visible. There is deliberately space left between the border and the matrix, to be able to better see the fine arrow-like patterns in **BA** datasets.

that D is indeed diagonal (note that although A is only taking a single pixel in the top-left corner, it also is diagonal). The Schur complement of A is:

$$\text{Schur}(A) \triangleq A - UD^{-1}U^T. \quad (8.2)$$

This can be used to solve the original system as:

$$(A - UD^{-1}U^T) \mathbf{x} = \mathbf{a} - UD^{-1}\mathbf{b}, \quad (8.3)$$

$$\mathbf{y} = D^{-1}(\mathbf{b} - U^T\mathbf{x}), \quad (8.4)$$

where the former is a smaller, more dense system that can be solved using a general linear solver and the latter is merely a matrix vector product. The advantage of this procedure is that inverting D amounts to inverting its individual diagonal blocks which is an embarrassingly parallel operation. Additionally, in the **BA** type problems, D contains the most of the rank of the system matrix so that a large part of the system is solved quickly.

The smaller dense system (8.3) is often referred to as the *reduced camera system* since it contains the camera poses. To solve it, several types of direct solvers have been applied in the literature. It is possible to use dense Cholesky or dense LDL^T decompositions². Densities of as high as 40% occur on e.g. the *Venice* dataset [106] (see Figure 8.1c). Sparse Cholesky solvers have shown about an order of magnitude speedups, especially on large systems and while using a good ordering. The fill-reducing orderings used for sparse Cholesky in **BA** implementations include Multiple Minimum Degree (**MMD**) [112], **AMD** [7] or even Reverse Cuthill-McKee (**RCM**) [37] (although most likely only in an attempt to point at the disadvantages of direct solvers). For perspective, dense Cholesky solver on **GPU** achieves up to two orders of magnitude speedup (including the data transfers) but is limited by the available memory.

While sparse LDL^T , **LU** or even **QR** seem like viable options, it is necessary to take the pivoting into the account: these factorizations are not implicitly numerically stable (unlike Cholesky) and may require row or column interchanges as the factorization progresses. These interchanges are typically implemented to improve the results numerically but ignore the fill-in they cause (as already discussed in Section 5.2.5).

Surprisingly, while using the Schur complement leads to reduction in computation time, it does not lead to reduction in complexity. For the *Venice* dataset, calculating the Cholesky factorization of Λ and solving for a single right hand side requires $25.432 \cdot 10^9$ and $248.347 \cdot 10^6$ **FLOPs**, respectively³. On the other hand, calculating the Schur complement and its Cholesky factorization takes $50.088 \cdot 10^9$ **FLOPs** and solving for a single r.h.s. takes $260.917 \cdot 10^6$ **FLOPs**. The situation is similar for the *Guildford Cathedral* and *Fast & Furious* ⁶⁴ datasets, which observe 67.03% and 31.49% increase in the operations count, respectively. On the other hand, using a serial implementation of Schur complement leads to speedups greater than

The RCM ordering is not a fill-reducing ordering but rather a bandwidth reducing one. It was developed in the 70's and there are much better orderings for direct methods today.

² In here, the D is a generic diagonal matrix, other than that in (8.2).

³ These figures were calculated by defining a custom numeric type which counts operations performed and making the **CXSparse** library use it, thus counting exact numbers of **FLOPs** in sparse matrix operations. The implementation is available as a part of the **SLAM++** library, at <http://sf.net/p/slam-plus-plus>.

⁴ Kindly provided by Double Negative, <http://www.dneg.com>.

3.5× in all three datasets, compared to the direct solution of normal equations via sparse block factorization.

This is because the operations used in Schur complement are simpler ones (for the most part only multiplications and additions) compared to the Cholesky factorization (which requires also a fair amount of divisions and square roots). However, the differences of the cost of these operations is diminished by the use of **SIMD** instruction sets which can often execute any kind of instruction in a single clock. The memory accesses are also more organized in Schur complement, making a better use of **CPU** cache. Additionally, matrix multiplication, block diagonal inverse and dense solving are all parallelizable, with much better scaling than sparse Cholesky factorization.

8.1 FINDING GOOD ORDERING

Linear solving using the Schur complement relies on D being diagonal, or rather block diagonal in the context of problems with multi-dimensional variables. As mentioned earlier, the graph theoretic algorithms useful for finding diagonal sections are the ones for finding bipartite graphs and for finding maximum independent sets. In the case a bipartite graph is found, ordering the variables in such a way that one set of independent variables resides in A and the other one in D yields a block-diagonal A and D with all the off-diagonal entries collected in U and U^T . If the problem at hand does not correspond to a bipartite graph, finding a maximum independent set and ordering the independent variables to reside in D and the rest of the variables in A yields another configuration which can be efficiently solved using Schur complement. There are efficient implementations of both these algorithms, e.g. the `igraph` [36] library⁵ implements [56] for finding maximal clique sets and [170] for finding maximal independent vertex sets. Here, the word *maximal* means that for a given clique (or equally an independent vertex set), no additional vertices can be added to it. However, *maximum* (or the greatest) independent vertex set is the one set which has the most vertices of all the maximal independent vertex sets in the graph. This is what is referred to as Maximum Independent Set (**MIS**).

In **BA** problems, an often used approach is ordering the 3D point variables to reside in D since they are independent (there are no observations of a structure point by another structure point) and the rest of the variables to reside in A . This is referred to as the *guided* ordering. As a side note about the implementation, block matrices can be elegantly taken advantage of – the blocks corresponding to the 3D points have unique size of 3×3 (both cameras and intrinsic parameters would

⁵ Can be found at <http://igraph.org/c/>.

 Algorithm 8.1: Finding Maximum Independent Clique Sets.

```

1: function MICS( $w, e$ )
Require:  $w = [w_1, \dots, w_n]$  is the vector of vertex weights.
Require:  $e = \{e_1 \dots e_m\}$  is a set of edges, where each edge  $e_i$  is a pair  $(j_i, k_i)$ .
2:    $C = \text{FINDCLIQUES}(e)$   $\triangleright$  Use e.g. algorithm of Eppstein et al. [56].
3:    $P = [\emptyset, \dots, \emptyset]$   $\triangleright P_v$  is a set of cliques containing vertex  $v$ .
4:   for each  $c$  in  $C$  do
5:      $w = \left[ w; \sum_i w_{c_i} \right]$   $\triangleright$  Clique weight is a sum of weights of its vertices.
6:     for each  $v$  in  $c$  do
7:        $P_v = P_v \cup c$ 
8:     end for
9:   end for
10:  for each  $c$  in  $C$  do
11:     $V_{\text{adj}} = c \cup \{v \mid \exists e_i = (v, u) \in e \wedge u \in c\}$   $\triangleright$  Vertices adjacent to clique  $c$ .
12:     $C_{\text{adj}} = \{P_v \mid v \in V_{\text{adj}}\}$   $\triangleright$  Cliques adjacent to clique  $c$ .
13:     $e = e \cup \{(c, v) \mid v \in V_{\text{adj}}\} \cup \{(c, d) \mid d \in C_{\text{adj}}\}$   $\triangleright$  Add new edges.
14:  end for
15:  Return  $\text{MAXINDEPENDENTSET}(e, w)$   $\triangleright$  Use e.g. Tsukiyama et al. [170].
16: end function

```

have blocks of higher dimensions) and the BA matrices can therefore be ordered based on the block structure only, rather than by passing variable type information to the linear solver (which works on a different level of abstraction entirely).

For landmark SLAM, the guided ordering is often a poor fit, since the landmarks often take up only a small fraction of the matrix rank. Consider the *Victoria Park* [133] dataset (described earlier in Section 6.2, Figure 6.2c), a 2D landmark SLAM dataset with 6969 poses and only 151 landmarks (1.44% of the rank, see the part of the matrix marked by the red square in Figure 8.2a). Note that although the top left part of the matrix appears diagonal, there are off-diagonal elements corresponding to the odometry links which connect the consecutive poses. Those make the matrix band diagonal and no longer easily invertible. A better result is obtained by finding a Maximum Independent Set (MIS) weighted by variable dimension which yields D that amounts to about 47.83% of the rank, see Figure 8.2b.

Unfortunately, not all graphs are so sparse and the MIS ordering does not always give such a good results (e.g. on the *10k* dataset, the MIS ordering yields the diagonal section with less than 20% of the rank). For that reason, a new ordering strategy was devised. The goal is to create a block diagonal section in D of the highest rank possible. To achieve that, the block diagonal does not need to be of the granularity of the individual variable blocks, but can contain greater blocks.

Since the algorithms are operating at the level of variables (or blocks) rather than elements, the weights are necessary to get the independent set with the largest number of elements.

Those correspond to the independent cliques in the original graph. The algorithm for finding the maximum (weighted) independent clique set is not implemented in the `igraph` library (or other library, to the best of our knowledge). The implementation is described concisely in [Algorithm 8.1](#). In the first part of the algorithm, the cliques are found. Then, the original graph is extended with the cliques and the relations to other vertices and cliques are computed, see [Figure 8.2c](#). In this way, the cliques can be treated as ordinary vertices and the weighted maximum independent set can be found using a standard algorithm.

While the performance of the maximal cliques algorithm is reasonable and typically takes only a few milliseconds, the maximal independent vertex sets algorithm is not practical for even small graphs, e.g. on the *Intel* graph (943 vertices, 0.52% nonzeros) it requires more than 120 GB of memory. Therefore, an *approximate* algorithm was devised, based on a simple first-fit scheme followed by iterative refinement. The Approximate Maximum Independent Clique Set ([AMICS](#)) ordering on *Victoria Park* yields D which takes 50.61% of the rank, see [Figure 8.2d](#).

8.2 INCREMENTAL SOLVING

Similarly to [SLAM](#), the [BA](#)-type problems are also often solved incrementally. This is needed to avoid divergence, especially due to poor prediction of camera parameters, which can lead to bad initialization of point positions and consequent camera poses quickly since the projection amplifies the error. Unlike [SLAM](#) where the update usually consisted of a handful of new observations and a single new pose, however, the rank of the updates is much bigger this time. For each new camera pose, thousands of points can be observed, many of them for the first time.

The goal is to describe how changes in Λ translate to changes in the Schur complement of A . Updates to D^{-1} are handled easily, as all the updated diagonal blocks in \hat{D} can be inverted individually and the rest does not change. It can be expected in practice that all four sections of Λ are going to change:

$$\begin{pmatrix} \hat{\Lambda}_{11} & \hat{\Lambda}_{12} \\ \hat{\Lambda}_{12}^\top & \hat{\Lambda}_{22} \end{pmatrix} = \begin{pmatrix} A & u \\ u^\top & D \end{pmatrix} + \begin{pmatrix} \Delta A & \Delta u \\ \Delta u^\top & \Delta D \end{pmatrix}. \quad (8.5)$$

This difference is nonzero only in the new or changing blocks, but $\Delta(D^{-1}) \neq (\Delta D)^{-1}$.

Taking the difference $\Delta(D^{-1}) = \hat{D}^{-1} - D^{-1}$ after the inverse, the update is:

$$\begin{aligned} \text{Schur}(\hat{A}) &= A + \Delta A - (u + \Delta u)(D^{-1} + \Delta(D^{-1}))(u^\top + \Delta u^\top) \\ &= A + \Delta A - (u + \Delta u)D^{-1}(u^\top + \Delta u^\top) - (u + \Delta u)\Delta(D^{-1})(u^\top + \Delta u^\top) \\ &= A + \Delta A - uD^{-1}u^\top - uD^{-1}\Delta u^\top - \Delta uD^{-1}(u^\top + \Delta u^\top) - \\ &\quad (u + \Delta u)\Delta(D^{-1})(u^\top + \Delta u^\top) \\ &= \text{Schur}(A) + \Delta A - uD^{-1}\Delta u^\top - \Delta uD^{-1}\hat{u}^\top - \hat{u}\Delta(D^{-1})\hat{u}^\top \end{aligned} \quad (8.6)$$

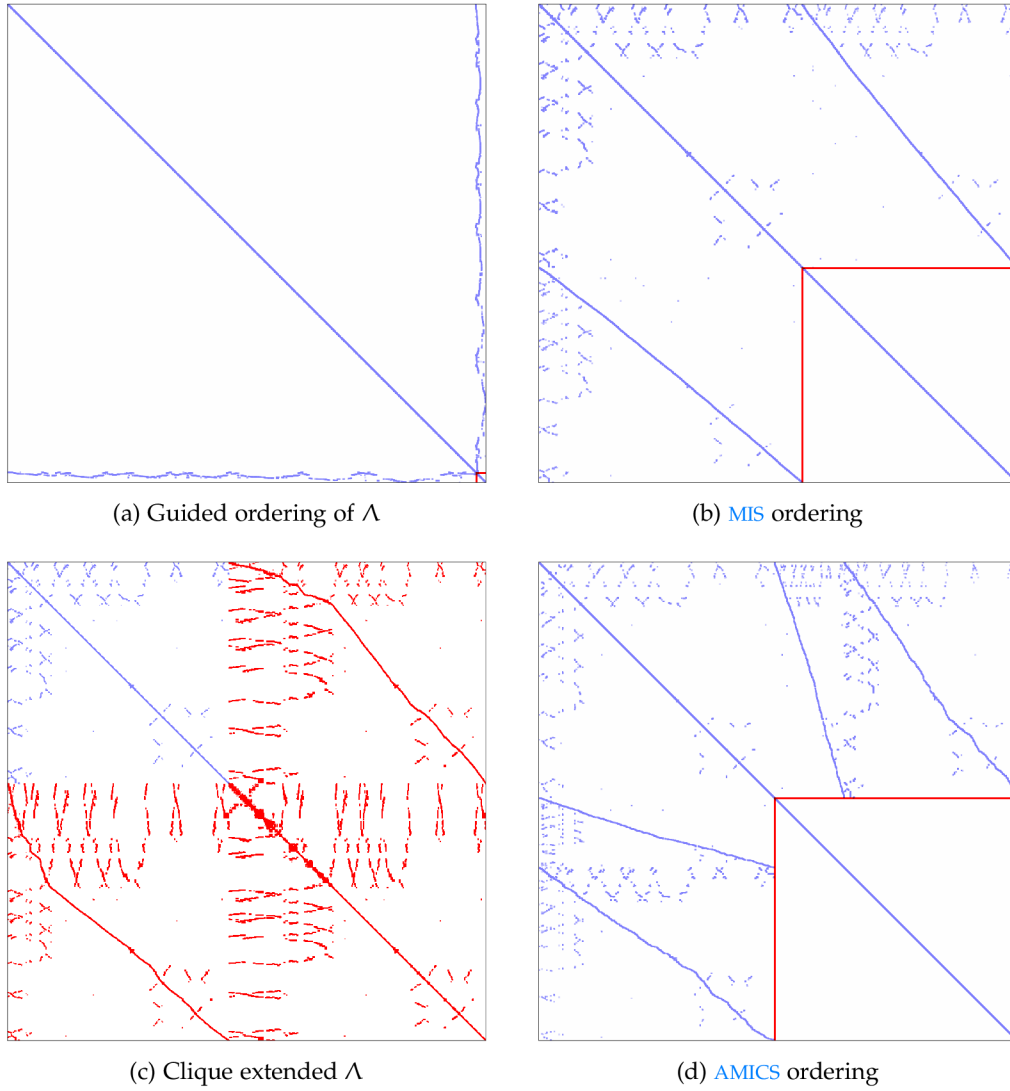


Figure 8.2: Finding Schur ordering for landmark SLAM, on the *Victoria Park* dataset.

and by taking advantage of symmetry:

$$D^{-1} = D^{-\top}, \quad (8.7)$$

$$(\mathbf{U}D^{-1}\Delta\mathbf{U}^{\top})^{\top} = (\Delta\mathbf{U}^{\top})^{\top}D^{-\top}\mathbf{U}^{\top} = \Delta\mathbf{U}D^{-1}\mathbf{U}^{\top}, \quad (8.8)$$

it is possible to further simplify (8.6) to:

$$\begin{aligned} \text{Schur}(\hat{\mathbf{A}}) &= \text{Schur}(\mathbf{A}) + \Delta\mathbf{A} - (\Delta\mathbf{U}D^{-1}\mathbf{U}^{\top})^{\top} - \Delta\mathbf{U}D^{-1}\hat{\mathbf{U}}^{\top} - \hat{\mathbf{U}}\Delta(D^{-1})\hat{\mathbf{U}}^{\top} \\ &= \text{Schur}(\mathbf{A}) + \Delta\mathbf{A} - (\Delta\mathbf{U}D^{-1}\hat{\mathbf{U}}^{\top} - \Delta\mathbf{U}D^{-1}\Delta\mathbf{U}^{\top})^{\top} - \Delta\mathbf{U}D^{-1}\hat{\mathbf{U}}^{\top} - \\ &\quad \hat{\mathbf{U}}\Delta(D^{-1})\hat{\mathbf{U}}^{\top} \\ &= \text{Schur}(\mathbf{A}) + \Delta\mathbf{A} - (\mathbf{E} - \mathbf{F}\Delta\mathbf{U}^{\top})^{\top} - \mathbf{E} - \hat{\mathbf{U}}\Delta(D^{-1})\hat{\mathbf{U}}^{\top}, \end{aligned} \quad (8.9)$$

with $\mathbf{E} \triangleq \mathbf{F}\hat{\mathbf{U}}^{\top}$ and $\mathbf{F} \triangleq \Delta\mathbf{U}D^{-1}$ being common subexpressions. Note that this way, each of the product terms contains at least a single matrix of low rank (either $\Delta\mathbf{U}$ or $\Delta(D^{-1})$) which limits the amount of computation and also only $\text{Schur}(\mathbf{A})$ and D^{-1} need to be stored from the previous step, limiting the required amount of memory for the incremental solver.

For the Venice dataset, this saves 83% of memory.

Due to the highly nonlinear nature of BA, the nonlinear solvers typically take some form of countermeasure to avoid local minima (e.g. as described in [Section 2.1.1](#)). By employing the Levenberg-Marquardt algorithm [125], a diagonal damping term λ is introduced, yielding a modified normal equation (2.13). This term does change during the solving, causing full-rank incremental updates. For that reason, the Dogleg algorithm [141, 25] is preferred for incremental solving.

8.2.1 Fluid Relinearization

By adding the cameras and points incrementally, the estimated state is always close to the optimal solution. This means that in the nonlinear steps, most of the variables will not change significantly. This is also affected by an appropriate parameterization of the problem, e.g. by the choice of the representation of the points and by the choice of global or local coordinate frames for representing the points and cameras.

Several possible point representations include *Euclidean* (each point is represented by the $[x, y, z]$ coordinates), *inverse depth* (each point is represented by $[\frac{x}{z}, \frac{y}{z}, \frac{1}{z}]$) or *inverse distance* [117] (overparameterized (u, v, w, q) where $q = \|[x, y, z]\|^{-1}$ is reciprocal Euclidean distance from the origin and $(u, v, w) = q(x, y, z)$ is a unit vector which points towards the point and is kept constant throughout the optimization process).

With that in mind, the points can be in the global coordinate space (which only really suits the Euclidean representation) or in the local coordinate frame of one of the cameras (which suits any of the parameterizations). Note that keeping the the points in a local coordinate frame changes the graph structure; the nonlinear measurement function $h_k(\cdot)$ for an observation by a camera θ_c of a point θ_p represented in local coordinate frame of a different camera θ_d now requires three arguments rather than two in the former case of a camera observing a point in the global coordinate frame. The cameras themselves can also be represented in a local coordinate frames of each other, which leads to chaining of the transformations and measurement functions with variable number of arguments. The choice of the camera relations becomes significant from the sparsity and computational requirements point of view.

In any case, a suitable representation can significantly reduce the magnitude of the steps of the nonlinear solver, while at the same time not increasing the density of the system too much. It is then possible to apply threshold to the δ vector in (2.7). This reduces the rank of the updates while reducing the convergence slightly. In case a sufficient number of iterations is available, the nonlinear solver will not diverge. This is intuitively illustrated on the Dogleg algorithm ([Section 2.1.1](#)) which

This is commonly referred to as the fluid relinearization.

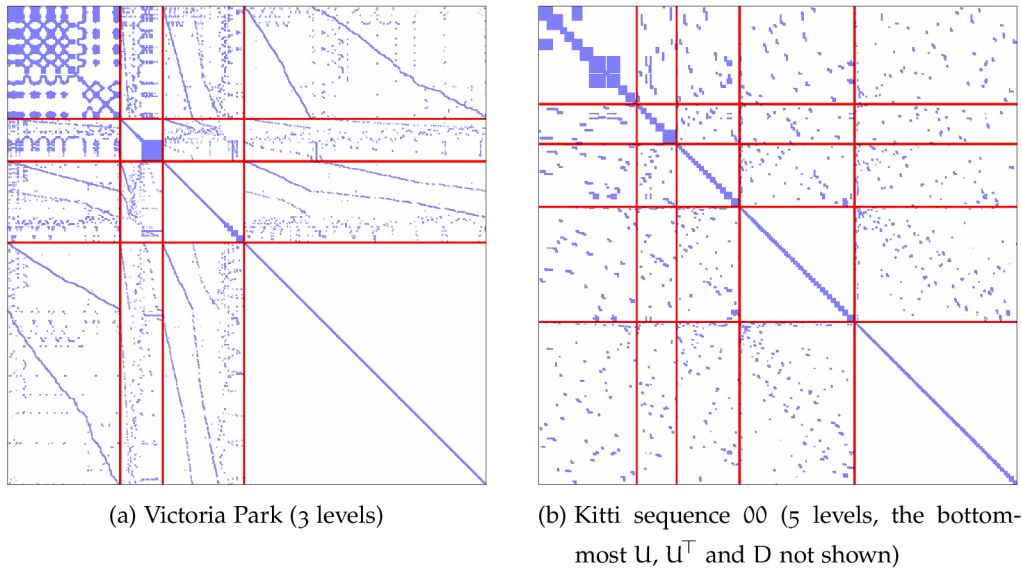


Figure 8.3: Examples of nested Schur complements using the [AMICS](#) ordering.

chooses step inside the trust region, a sphere of radius Δ . In case some components of this step are forced to zero by the threshold, the solution still stays inside the trust region. Ultimately, there is a trade-off between the sparsity of the updates which makes the updates faster, and the number of extra iterations taken which makes the solver slower again. Thresholds of magnitude about 10^{-5} have proven to provide a reasonable balance.

8.3 NESTED SCHUR COMPLEMENT

Another interesting option of Schur complement is the possibility to create nested Schur complements. In (8.3), the reduced camera system needs to be solved. It can be readily solved using Cholesky factorization as described before, but in case it is sparse enough, it can be solved using another Schur complement, yielding a nested Schur complement method. Nesting the Schur complements is only beneficial in case the reduced camera system needs to be solved using a dense solver (e.g. a solver parallelized on a [GPU](#)) and still contains too many nonzero entries or is too large to fit into the memory at once.

High sparsity is typically not a case of [BA](#) problems where the reconstructed object is observed in its entirety by the majority of the cameras, but occurs in cases when the camera moves forward in exploratory mode and only rarely re-observes small parts of the scene. Size is a hard limit though; for a 4 GB memory budget, the dense reduced camera system can hold only up to 3861 6D camera poses (assuming the internal parameters are either known and not optimized, or identical for all the frames – otherwise this figure would be even lower). This is often not sufficient, e.g. *Kitti sequence 00* [63] comprises over 4500 poses.

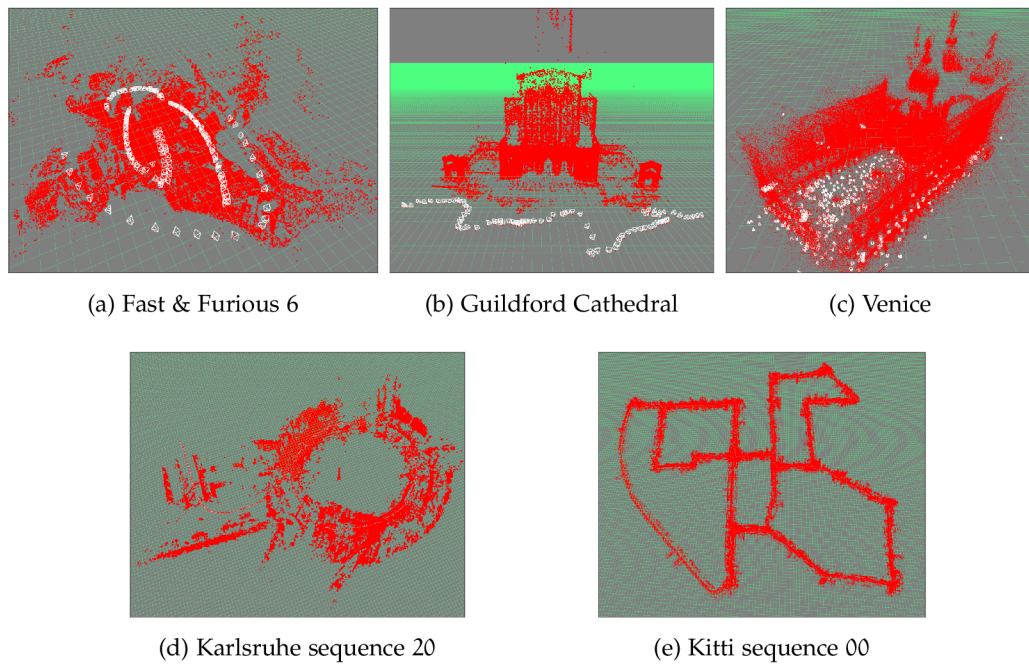


Figure 8.4: The Bundle Adjustment (BA) datasets used in the evaluations. The top row are datasets focused on 3D reconstruction, the datasets on the bottom row are visual odometry benchmarks.

8.4 EXPERIMENTAL EVALUATION

The experimental evaluations were performed on several datasets which can be seen in Figure 8.4. *Fast & Furious 6* is a bundle adjustment dataset comprising of 160 high-resolution DSLR stills of an open landscape and a highway bridge in Gran Canaria⁶. The images were captured from a helicopter for production of special effects in a chase sequence in the movie of the same name. The dataset was kindly provided by Double Negative Visual Effects⁷. *Guildford Cathedral* is another bundle adjustment sequence made up of 92 DSLR stills, scanning the front facade of the Guildford Cathedral (Surrey, London) in approximately right to left translational manner. The dataset is freely available (upon request) at <http://cvssp.org/impact/>. *Venice* is a standard bundle adjustment dataset [106] created from an internet collection of 871 photos of a courtyard adjacent to the San Marco square in Venice, Italy.

Karlsruhe sequence 20 [62] is visual odometry benchmark, processed with a stereo structure from motion pipeline. Although the observation model of the stereo BA is slightly different from the monocular one, the variable representations and the corresponding Jacobian matrices have exactly the same structure and dimensions. The images were taken with a camera mounted on top of a car and this sequence

⁶ GPS coordinates of the approximate center of the dataset are 28.1396417N, 15.5973228W.

⁷ <http://www.dneg.com/>

Table 8.1: Linear solving performance on the standard BA datasets, the best times in bold.

Dataset	Fast & Furious 6	Guildford Cathedral	Venice	Karlsruhe seq. 20	Kitti seq. 00
direct- Λ -CS	2.243	2.583	66.309	11.983	4.995
direct- Λ -CM	1.992	1.726	38.272	4.943	4.583
direct- Λ -BC	2.250	1.954	47.382	4.456	3.469
Schur-BC	0.623	0.544	14.254	5.135	3.261
Schur-GPU K20m	0.612	0.536	6.472	1.610	2.363
Schur-GPU K40c	0.361	0.446	5.057	1.516	1.652
Ordering (levels)	guided	guided	guided	guided	AMICS (5)

has 967 of them. A similar dataset, *Kitti sequence 00* of the newer vision benchmark suite by the same authors [63] is a representative of a large problem, with its 4541 camera poses.

Some of the tests were performed on an Intel Core i5 CPU 661 with 8 GB of RAM and running at 3.33 GHz, equipped with the NVIDIA Tesla K40c GPU. Additionally, some tests were performed on a machine with a pair of Intel Xeon E5-2470 CPUs running at 2.30 GHz and sharing 96 GB of RAM, equipped with a single NVIDIA Tesla K20m GPU. During the tests, the computers were not running any time-consuming processes in the background. Each test was run several times and the average time was calculated in order to avoid measurement errors. Note that the Xeon CPUs have a turbo boost feature that 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.

The GPUs were employed for dense solving using CULA⁸ and for block diagonal inverse, using Cholesky and LU decompositions, respectively. A serial CPU implementation of the sparse block matrix multiplications was employed in (8.2) or (8.3) as it proved to be faster than the off-the-shelf GPU routines. This further demonstrates the efficiency of the block schemes.

The batch solving was evaluated using direct solution of the normal equation, using CSpase, Cholmod and block Cholesky. These times serve as the baseline and are compared to the Schur complement methods using block Cholesky on CPU and the dense Cholesky on GPU. The times are in Table 8.1. From the direct solvers, CSpase is the slowest. Cholmod is the fastest on the first three datasets which are more connected and the supernodal method is more advantageous. Block Chol-

Note that all of the processing times would be lower with turbo boost enabled.

⁸ A readily available GPU accelerated LAPACK implementation written in CUDA, can be obtained from <http://www.culatools.com/>,

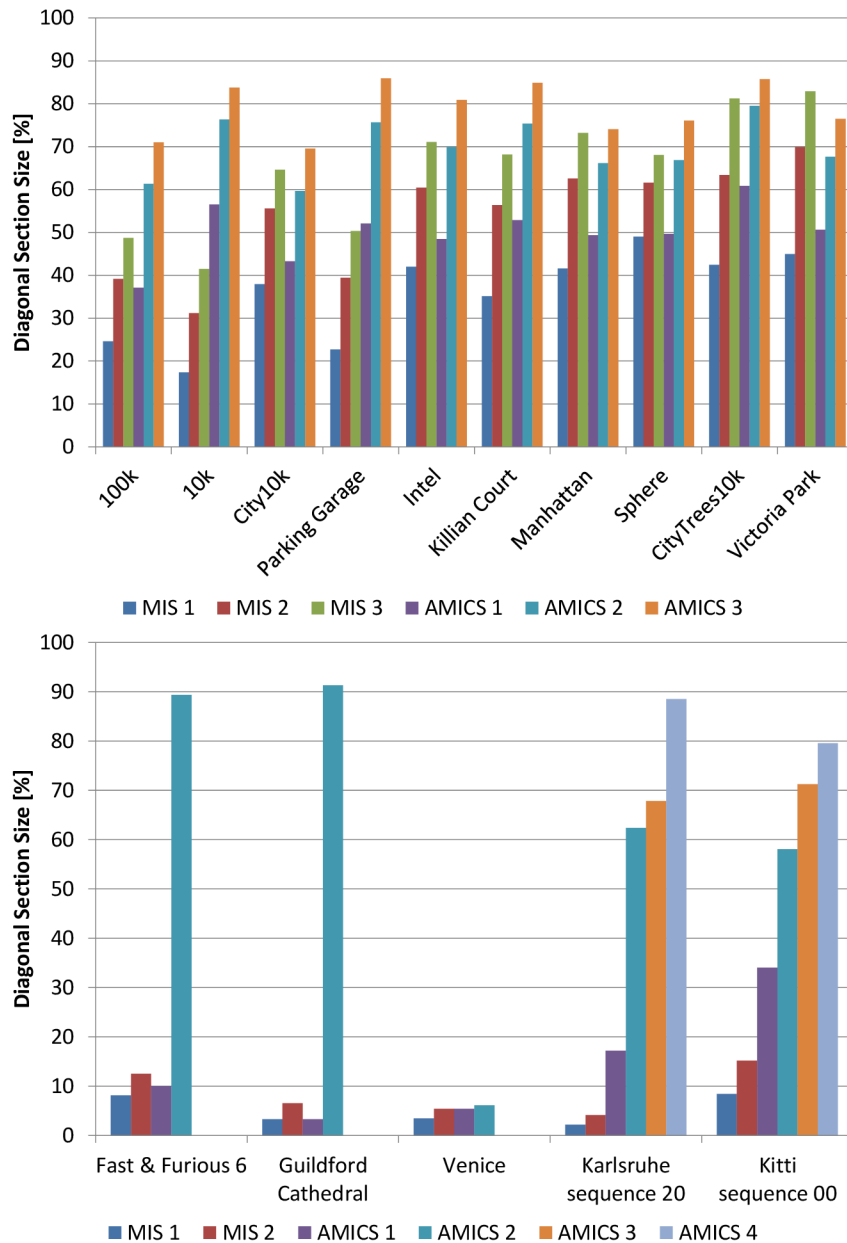


Figure 8.5: Evaluation of nested Schur orderings for up to 3-level nested Schur complements on standard [SLAM](#) and up to 4-level on the Schur complements of the [BA](#) datasets (the bigger the percentage, the better – it means smaller, denser reduced camera system). The number after the ordering acronym indicates the nesting level.

esky is the fastest on the last two datasets which are much bigger and less connected, yielding fewer supernodes. From the Schur solvers, the [GPU](#) solution on Tesla K40 is the fastest. The Schur complement is always faster than the direct factorization, even on the [CPU](#), with the exception of *Karlsruhe sequence 20* dataset which is already very sparse (but still, the [GPU](#) accelerated solution gains a substantial speedup). Note that these times do not involve the tasks of the nonlinear solver which would be the same for all of the approaches, such as assembling the system matrix and calculating the derivatives.

The ordering algorithms for Schur complement were also evaluated. In [Figure 8.5](#), different ordering strategies are compared in terms of the size of the diagonal section D relative to the size of the entire system (for nested Schur complements, these sizes are summed up). In the top portion of the figure, ordering performance on the [SLAM](#) datasets is compared. Since these datasets are typically very sparse, the [MIS](#) ordering is able to improve by nesting. However, the [AMICS](#) ordering yields much better results.

On the [BA](#) datasets in the lower portion of the figure, rather than evaluating on the full matrix, the orderings are evaluated on the Schur complement obtained by the guided ordering (which is coincidentally the same one as the [MIS](#) and also the [AMICS](#) since the landmarks are the largest independent set and there are no cliques). These Schur complements are about two orders of magnitude more dense than the [SLAM](#) systems, which reflects poorly on the [MIS](#) orderings.

For the first three datasets, [AMICS](#) ordering splits the Schur complement in almost completely dense block-diagonal section and another completely dense Schur complement. The difference between *Fast & Furious 6* or *Guildford Cathedral* and *Venice* is that in the former two the most of the rank ends up in the diagonal section whereas in *Venice*, it ends up in the top-level Schur complement. For the last two datasets which are more sparse, [AMICS](#) gradually improves with nesting. However, size is not everything, as reflected in [Figure 8.6](#). Here, in the first two datasets all the orderings come out more or less the same. In *Venice*, the nested [MIS](#) is surprisingly slightly faster while the [AMICS](#) are slightly slower. In *Karlsruhe sequence 20*, [AMICS](#) yield poor performance compared to simpler orderings. This is because the system is already sufficiently dense after the first level. On the other hand, on *Kitti sequence 00*, the Schur complement is quite large and a few nestings are required to fit the problem into the [GPU](#) memory. Here is where [AMICS](#) triumphs.

For the evaluation of the incremental solving, the [BA](#) datasets which are in graph format were preprocessed by an external tool⁹. First, the variables were reordered so that the cameras go in a sequence and the landmarks are introduced once observed by at least two cameras (at that point they could have been triangulated). Additionally, for the solver to determine the points where to optimize, frame boundaries markers are inserted. Note that this preprocessing would be unnecessary if the solver was connected to a vision pipeline – it is only needed when processing datasets where the variables were reordered and the frame boundaries were lost or perhaps a linear camera sequence never existed in the first place. The results are in [Table 8.2](#) and involve the full solution of the nonlinear system. Note that the *Kitti sequence 00* was not included in this evaluation due to its size – the

Ultimately, the performance depends on the size of the final dense factor which [AMICS](#) excels at minimizing, but also on the cost of forming the nested Schur complement. This leads to a tradeoff problem.

⁹This script can be found under `scripts/incremental_BA` in the `SLAM++` library, at <http://sf.net/p/slam-plus-plus/>.

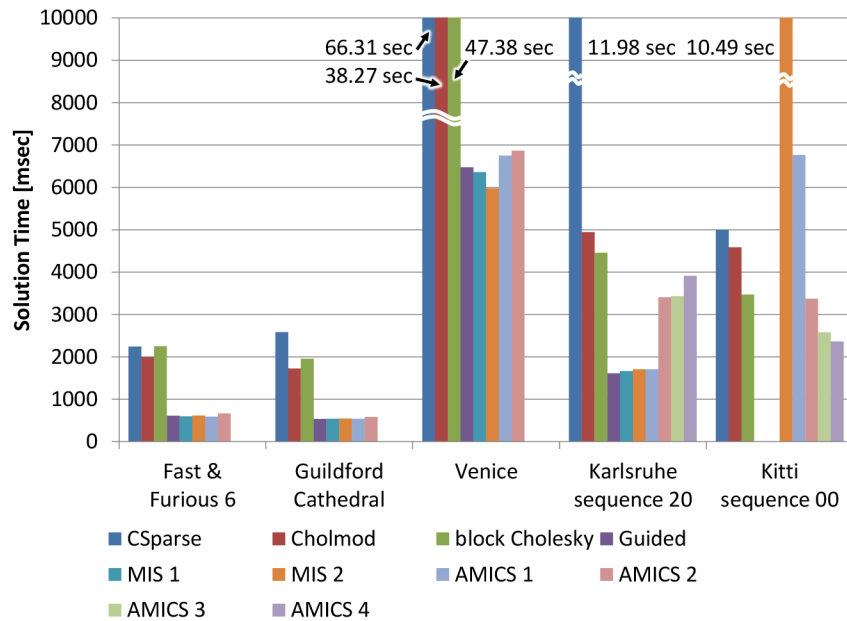


Figure 8.6: Comparison of the effects of ordering on the linear solving on the standard BA datasets. All Schur complement times were obtained using the K20m GPU. The missing times for the *Kitti sequence 00* dataset were caused by insufficient amount of available memory.

problem is in the ordering. Unlike guided ordering, the orderings based on Maximum Independent Set (MIS) are not stable in the sense that in one step, a variable may be a part of MIS but in the next step, a different independent set containing different variables may become the maximum one. This would require variable migration between the diagonal section and the reduced camera system which in itself is feasible, but the volume of the variables is practically unlimited. For that reason, incremental Schur complement was only evaluated with the guided ordering.

8.5 CHAPTER SUMMARY

This chapter demonstrated solving Bundle Adjustment (BA) problems using the Schur complement methods and also using dense GPU acceleration primitives. Notably, the BA problems are suitable for GPU acceleration, reaching good speedups while only using off-the-shelf dense kernels. Unfortunately, there are some limitations that require further attention. The matrix-matrix product on GPU is slower than the SSE-accelerated serial sparse block matrix implementation described in Section 5.2.4 and becomes the main bottleneck. More efficient algorithms are needed. Also, the dense solving on the GPU scales very well, however it is severely limited by the available memory. One option is using the Schur complement to reduce the size of the dense system, another orthogonal approach would be imple-

Table 8.2: Incremental nonlinear solving performance on the standard BA datasets, the best times are in bold (CS is CSparse, CM is Cholmod, BC is block Cholesky and K20m and K40c refer to GPU models).

Dataset	Fast & Furious 6	Guildford Cathedral	Venice	Karlsruhe seq. 20
allBatch- Λ -CS	359.962	171.860	16179.340	3279.406
allBatch- Λ -CM	363.267	181.735	11296.632	2029.337
allBatch- Λ -BC	334.708	165.988	12949.737	1475.498
Schur-BC	220.304	103.874	4339.209	1094.002
Schur-GPU K20m	221.991	102.105	2897.663	605.795
Schur-GPU K40c	43.034	62.293	1797.362	402.582
incSchur-BC	198.335	90.103	2701.169	775.428
incSchur-GPU K20m	197.285	90.913	1249.735	314.500
incSchur-GPU K40c	31.868	56.255	945.488	177.476

menting either banded or jagged-diagonal Cholesky factorization on GPU which would inherit some of the benefits of the dense algorithms but would permit larger problems to be handled.

It was demonstrated how to use graph theoretic algorithms to calculate orderings and a novel Maximum Independent Clique Set (MICS) / AMICS orderings were proposed and evaluated. The benefit of having larger diagonal section is having a larger portion of the computation spent in diagonal inversion and matrix multiplication and having a smaller denser Schur complement. Additionally, it enables the use of a GPU even on very large datasets. There are, however, some improvements that could be made – the ordering of the variables inside the cliques is now arbitrary. It would be possible to order the cliques in such a way that the number of FLOPs is reduced or e.g. so that the off-diagonal blocks (U and U^T in (8.2)) have their bandwidth reduced in order to increase cache coherency in matrix-matrix and matrix-vector products in (8.3) and (8.4).

Equations for incremental solving using Schur complement were also demonstrated and evaluated, yielding promising results. There are some limitations, however. Since large portions of the Schur complement itself change in the incremental updates, it is not possible to employ nested Schur complement in incremental solving (although it would still be possible to compute the bottom-most level incrementally and the rest of the levels in batch mode). An unfortunate limitation is the inability to use the AMICS ordering in the incremental setting due to independent set instability, as explained before.

COVARIANCE RECOVERY

The existing incremental [NLS](#) solutions provide fast and accurate estimations of the mean state vector, for example the mean position of the robot and of the features in the environment. However, in real applications, the uncertainty of the estimation plays an important role. This is given by the covariance matrix, which generalizes the notion of variance to multiple dimensions. In particular, the marginal covariances, which encode the uncertainties between a subset of variables, are required in many applications.

Data association is the problem of associating current observations with previous ones, and it is the key to reduce the uncertainty in [SLAM](#). Finding those associations becomes very expensive for large problems, nevertheless it can be simplified when the uncertainties of the estimates are known. Joint-compatibility tests in the case of landmark [SLAM](#) [129, 93] or estimation of possible relative displacement between poses in pose [SLAM](#) [89] are all based on recovering the marginal covariances. [Figure 9.1](#) shows how the data association problem can be restricted to only a small set of sensor registration indicated by the gray links between the current pose of the robot and close poses already visited.

Information theoretic measures, such as mutual information, are also computed using the marginal covariances. This allows for principled ways to reduce the complexity of the [SLAM](#) problem by selecting only the informative measurements [89] or to plan reliable paths with the least probability of becoming lost [171]. In computer vision, the mutual information is used in online systems to compute the most appropriate actions for feature selection [43] or in *active vision* to guide efficient tracking and image processing. It is also used in reducing the uncertainty in real-time monocular [SLAM](#) [172] and in *active matching* [79] of image feature. A problem related to active vision is the *next best view* for 3D reconstruction where the trace of the camera covariance matrix is used to select the images that will reduce the uncertainty in the reconstruction [80].

3D reconstruction has a wide variety of applications in computer graphics, robotics or digital cinema production, among others. Most of the existing 3D reconstruction frameworks only recover the *mean* of the reconstructed geometry. However, variance is the natural choice of estimate quality indicator, see [Figure 9.2](#) for an example of such use.

Even though recovering the mean of the estimate in the [BA](#) problems is relatively simple even at large scale, as documented by the previous chapter, recovering its

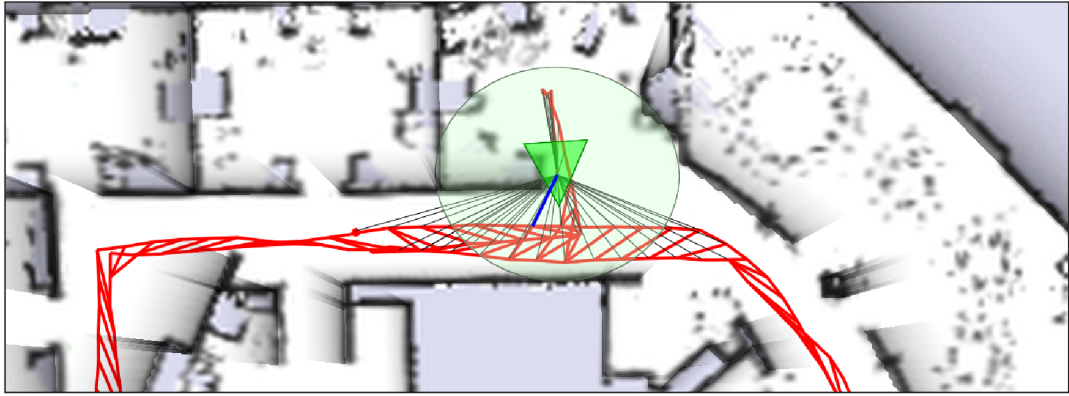


Figure 9.1: Distance-based candidates for data association calculated using the marginal covariances (95% confidence interval shown in green), on the *Intel* dataset.

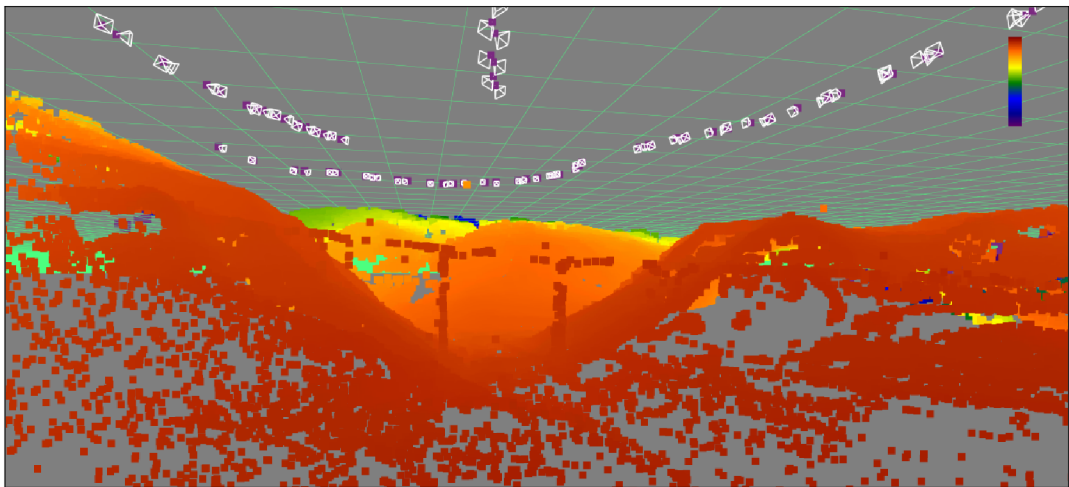


Figure 9.2: Marginal covariances used as a quality estimate of 3D reconstruction of the bridge sequence from the *Fast & Furious 6* dataset, displayed in false colors (orange means high confidence, blue – low confidence). Data courtesy of Double Negative Visual Effects.

covariance is significantly more difficult. One of the problems is that while the system matrix is sparse and can get very large, its inverse is completely dense and the memory footprint of maintaining such a matrix would be prohibitive, easily reaching hundreds of GB. Fortunately, for quality assurance and many other applications, only certain parts of the inverse are of interest – especially its block diagonal. Still, the problem of the computational complexity remains, which is the likely reason this problem was not widely addressed before.

9.1 RELATED WORK

While the covariances are explicit in Kalman filters and can also be easily recovered in information filters, which used to be commonplace in [SLAM](#) implementations, filtering is not widely used in [BA](#) or the 3D reconstruction problems in general,

since it is less efficient [160] and cannot take advantage of the various sparsity optimizations described in the previous chapters. In addition, there is a strict limit of the system size for which a dense matrix can be kept in RAM of today's systems, which would only allow solving moderate size problems.

Recovering the covariances in the context of nonlinear least squares is more difficult than in filtering. Thrun [165] proposes the use of so-called Markov blankets to approximate covariances of the poses of a robot. These are sub-blocks in an inverse of a smaller matrix that corresponds to the pose in question and the adjacent landmarks. It has been shown that those estimates are over-confident.

In [57], visual mapping of the sunken RMS Titanic is discussed and both the estimate and its covariance is recovered. The covariance is maintained incrementally: first, the covariances of the newly introduced variables is calculated by solving for the corresponding columns of the inverse system matrix. This column matrix is then fed to a bank of Kalman filters which update the covariances of the other variables.

An exact method for sparse covariance recovery was proposed in [93]. It is based on a recursive formula [18, 68], which calculates any covariance elements on demand from other covariance elements and elements of the Cholesky factorization of the system matrix. The downside of these methods when dealing with incremental online SLAM is the inability to take advantage of the incremental processing. The downside when dealing with BA problems is the need to calculate Cholesky factorization of the entire system, rather than to reuse the Schur complement and its factorization.

The authors of [142] proposed a covariance factorization for calculating linearized updates to the covariance matrix over arbitrary number of planning decision steps in a partially observable Markov decision process (POMDP). The method uses matrix inversion lemmas to efficiently calculate the updates. The idea of using factorizations for calculating inversion update is not new, though. A discussion of applications of the Sherman-Morrison and Woodbury formulas is presented in [77]. Specifically, it states the usefulness of these formula for updating the matrix inversion after small-rank modifications, where the rank is kept low enough to allow faster updates than actually calculating the inverse. In this chapter we propose a new update strategy which confirms this conclusion, but it has more practical application.

9.2 RECOVERING COVARIANCE IN GENERAL NLS PROBLEMS

When using MLE in real, online applications, the recovery of the uncertainty of the estimate, *the covariance*, can become a computational bottleneck. The calculation of

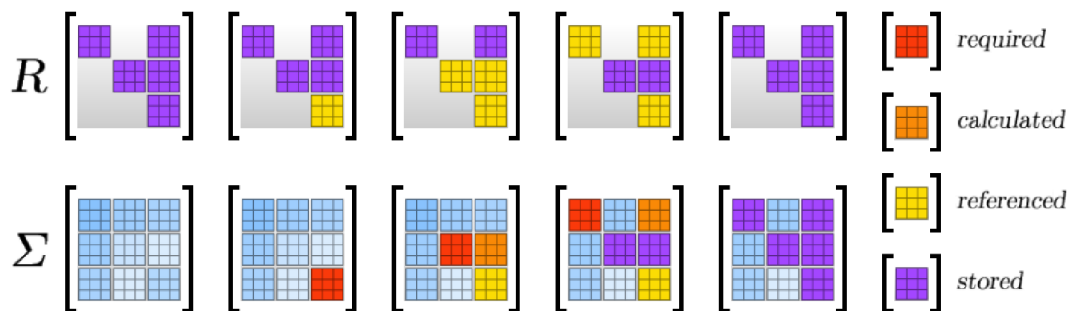


Figure 9.3: Recovering the diagonal of the covariance matrix using the recursive formula.

the covariance amounts to inverting the information matrix, $\Sigma = \Lambda^{-1}$, where the resulting block matrix Σ is no longer sparse. In here, each block $\Sigma_{i,j}$ corresponds to covariance between the individual variables θ_i and θ_j .

Operating on dense matrices is unwanted, especially in the case of large size matrix such as Σ . Fortunately, most of the applications require only a few block elements of the covariance matrix, eliminating the need of recovering the full Σ . In general, the elements of interest are the block diagonal and the block column corresponding to the last pose. Some other applications only require a few block diagonal and off-diagonal block elements. In [93], it was shown how specific elements from the covariance matrix can be efficiently calculated from the \mathbf{R} factor by applying the recursive formula:

$$\Sigma_{i,i} = \frac{1}{\mathbf{R}_{i,i}} \left(\frac{1}{\mathbf{R}_{i,i}} - \sum_{k=i+1, \mathbf{R}_{i,k} \neq 0}^n \mathbf{R}_{i,k} \Sigma_{k,i} \right), \quad (9.1)$$

$$\Sigma_{i,j} = \frac{1}{\mathbf{R}_{i,i}} \left(\sum_{k=i+1, \mathbf{R}_{i,k} \neq 0}^j \mathbf{R}_{i,k} \Sigma_{k,j} - \sum_{k=j+1, \mathbf{R}_{i,k} \neq 0}^n \mathbf{R}_{i,k} \Sigma_{j,k} \right). \quad (9.2)$$

Note that above, the computations are carried out by blocks; the numerical result is the same as if computed by elements but the calculation can be performed more efficiently. In case that \mathbf{R} is sparse, the formulas above can be used to compute the blocks of Σ at the positions of nonzero blocks in \mathbf{R} quickly [18]. To compute multiple blocks of the covariance matrix, such as the whole block diagonal, these formulas are efficient, provided all the intermediate results are stored. Figure 9.3¹ shows which elements need to be calculated for a specific block diagonal element.

9.2.1 Incremental Update of the Covariance Matrix

In Chapter 7, it was mentioned that most of the algorithmic speedups can be applied in case the linearization point is kept the same. Then, the contribution of

¹ An insightful animation of the covariance recovery, along with explanatory comments, is available online at <http://slam-plus-plus.sf.net/cov/>

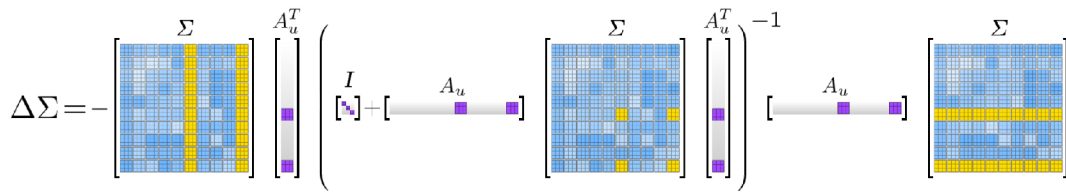


Figure 9.4: Sparsity patterns involved in covariance update calculation.

every new measurement can be easily integrated into the current system matrix Λ by a simple addition (see (6.3)), but things get complicated when the covariance is required:

$$\hat{\Sigma} = (\Lambda^\top \Lambda + A_u^\top A_u)^{-1} = (\Lambda + \Omega)^{-1}. \quad (9.3)$$

By applying the Woodbury formula, the above inverse can be written in terms of the previous covariance matrix:

$$\begin{aligned} \hat{\Sigma} &= \Lambda^{-1} - \Lambda^{-1} A_u^\top (I + A_u \Lambda^{-1} A_u^\top)^{-1} A_u \Lambda^{-1}, \\ \hat{\Sigma} &= \Sigma - \Sigma A_u^\top (I + A_u \Sigma A_u^\top)^{-1} A_u \Sigma. \end{aligned} \quad (9.4)$$

This shows that, in contrast to the information matrix which is additive, the covariance is subtractive:

$$\hat{\Sigma} = \Sigma + \Delta \Sigma, \quad \Delta \Sigma = -\Sigma A_u^\top (I + A_u \Sigma A_u^\top)^{-1} A_u \Sigma. \quad (9.5)$$

In *SLAM*, for example, this is easy to understand: a new measurement adds information to the system and *reduces* the uncertainty. It is important to mention that the size of the matrix to be inverted, $S \triangleq I + A_u \Sigma A_u^\top$, is very small compared to the system size. More precisely², the size of S , $m_u \times m_u$ with $m_u \ll m$, corresponds to the measurements involved in the update. For the simple case of a single measurement of a given *DOF*, $m_u = \text{DOF}$, regardless of the number of variables involved or their respective *DOFs*. Furthermore, due to the fact that A_u is very sparse, the computation of S can be performed very efficiently. The complex update in (9.5) becomes a simple block vector multiplication:

$$\Delta \Sigma = -\mathbf{B} \mathbf{S}^{-1} \mathbf{B}^\top \quad \text{where} \quad \mathbf{B} = \Sigma A_u^\top \quad \text{is a block vector.} \quad (9.6)$$

Due to the sparsity of the A_u , only few elements of the full Σ matrix are referenced, in particular only the block rows corresponding to the variables involved in the update. A simple example where the update involves two variables, is shown in [Figure 9.4](#). Furthermore, the size of \mathbf{B} is $n \times m_u$, but the product in (9.6) is a full matrix. Therefore the computation of the entire $\Delta \Sigma$ is prohibitive. We mentioned

² Assuming no new variables were added by the update, A is $m \times n$, A_u is $m_u \times n$ and Λ is a $n \times n$ matrix.

above that only some elements of the covariance are needed in the applications. For a single block, $\hat{\Sigma}_{i,j}$, the update can be easily calculated as:

$$\Delta \Sigma_{i,j} = -\mathbf{B}_i \mathbf{S}^{-1} \mathbf{B}_j^\top, \quad (9.7)$$

where \mathbf{B}_i and \mathbf{B}_j are block rows of \mathbf{B} of size of the update and the DOFs of the variables i and j ($\text{DOF}_i \times m_u$ and $m_u \times \text{DOF}_j$, respectively). A similar formulation of the covariance update was used in [89] in the context of filtering SLAM. In there, the marginal covariance of the variables were used to facilitate data association and graph sparsification using information theory measures.

The storage of the dense matrix Σ must be avoided. Only the blocks required by the application (for instance only the diagonal of Σ) are stored in a sparse block matrix. However, in order to compute the update in (9.6) or in (9.7), other elements of Σ are needed (the block columns, corresponding to the variables \mathbf{v} , involved in the update). Those are obtained by solving the system:

$$\Lambda \Sigma_{*,\mathbf{v}} = \mathbf{I}_{*,\mathbf{v}} \quad \text{or} \quad \mathbf{R} \Sigma_{*,\mathbf{v}} = \mathbf{R}^{-\top} \mathbf{I}_{*,\mathbf{v}}, \quad (9.8)$$

where \mathbf{I} is an identity matrix of the same size as \mathbf{R} and $\mathbf{I}_{*,\mathbf{v}}$ is a sparse block matrix containing only the block columns corresponding to the variables involved in the update. The complexity of this calculation is directly proportional to the sum of DOFs of the variables, involved in the update. For sparse \mathbf{R} with n_{nz} nonzero elements, calculating a single (elementwise) column of $\Sigma_{*,\mathbf{v}}$ by forward and back substitution amounts to $O(2n_{\text{nz}})$.

9.2.2 Incremental Downdate of the Covariance Matrix

Although very attractive, updating Σ as shown above sometimes becomes impractical to implement. In general, the covariances are calculated periodically, after the system was updated, which happens after one or several steps. In this case the Λ or \mathbf{R} are not available anymore, as they were replaced by $\hat{\Lambda}$ and $\hat{\mathbf{R}}$, respectively. Similarly to (9.3), one can downdate $\hat{\Lambda}$ to obtain Σ :

$$\Sigma = (\hat{\Lambda} - \mathbf{A}_u^\top \mathbf{A}_u)^{-1}. \quad (9.9)$$

Following the same scheme as in (9.4), $\Delta \Sigma$ can be now written in terms of $\hat{\Sigma}$:

$$\Delta \Sigma = +\hat{\Sigma} \mathbf{A}_u^\top (\mathbf{I} - \mathbf{A}_u \hat{\Sigma} \mathbf{A}_u^\top)^{-1} \mathbf{A}_u \hat{\Sigma}, \quad (9.10)$$

Defining $\mathbf{U} \triangleq \mathbf{I} - \mathbf{A}_u \hat{\Sigma} \mathbf{A}_u^\top$, which is a small size matrix similar to \mathbf{S} , (9.10) becomes:

$$\Delta \Sigma = \hat{\mathbf{B}} \mathbf{U}^{-1} \hat{\mathbf{B}}^\top \quad \text{with} \quad \hat{\mathbf{B}} = \hat{\Sigma} \mathbf{A}_u^\top, \quad (9.11)$$

while avoiding forming full $\hat{\Sigma}$ and only calculating the columns referenced in the above product, $\hat{\Sigma}_{*,\mathbf{v}}$. Those are easily obtained by solving $\hat{\Lambda} \hat{\Sigma}_{*,\mathbf{v}}^\top = \mathbf{I}_{*,\mathbf{v}}$ or $\hat{\mathbf{R}} \hat{\Sigma}_{*,\mathbf{v}} = \hat{\mathbf{R}}^{-\top} \mathbf{I}_{*,\mathbf{v}}$, much like in (9.8).

This allows us to update the covariance at any step from the current $\hat{\Lambda}$ or $\hat{\mathbf{R}}$ and a small A_u , instead of having to bookkeep the much larger Λ or \mathbf{R} alongside the updated $\hat{\Lambda}$ or $\hat{\mathbf{R}}$. Also, it is not mandatory to update Σ at each step: to perform update to Σ over several steps, A_u will simply contain all the measurements since Σ was calculated.

9.2.3 Alternative Update or Dwndate Formulation

It is also possible to apply the Woodbury formula slightly differently than in the previous subsections. Recall the Kailath variant of the Woodbury formula (see e.g. [17], page 153):

$$(X \pm YZ)^{-1} = X^{-1} \mp X^{-1}Y(I \pm ZX^{-1}Y)^{-1}ZX^{-1}. \tag{9.12}$$

In (9.10), the substitution was $X \leftarrow \hat{\Lambda}$, $Y \leftarrow A_u^\top$ and $Z \leftarrow A_u$ while it is also possible to set $Y \leftarrow I$ and $Z \leftarrow A_u^\top A_u = \Omega$. Note that an equivalent modification could have been applied to (9.4) as well. An update using the Ω matrix can thus be obtained, saving some computation (only a single product of $\hat{\Sigma}\Omega$ is needed, compared to two products $A_u\hat{\Sigma}$ and $A_u\hat{\Sigma}A_u^\top$ in (9.10)):

$$\Delta\Sigma = +\hat{\Sigma}\Omega(I - \hat{\Sigma}\Omega)^{-1}\hat{\Sigma}. \tag{9.13}$$

It is also possible to take advantage of the sparsity of the update and write $V \triangleq I - \hat{\Sigma}_{v,v}\Omega_{v,v}$, with Ω being a block version of Ω . It is worth noting that both $\hat{\Sigma}_{v,v}$ and $\Omega_{v,v}$ are full matrices and it is possible to use fast dense calculation. V is a $n_u \times n_u$ matrix where n_u is the sum of DOFs of variables in v . Although not universally true, usually $n_u \approx n_u$. This lets us transform (9.13) to:

$$\Delta\Sigma = \hat{\Sigma}_{*,v} \cdot C \quad \text{with} \quad C = \Omega_{v,v}V^{-1}\hat{\Sigma}_{*,v}^\top, \tag{9.14}$$

where C is a $n_u \times n$ matrix. Any element of the updated covariance matrix can be recovered as $\Delta\hat{\Sigma}_{i,j} = \hat{\Sigma}_{i,v} \cdot C_{*,j}$.

9.2.4 The Algorithm

In this chapter, an efficient algorithm for online recovery of the marginal covariances is proposed. Based on whether or not the linearization point changed, the algorithm has two branches: a) calculates sparse elements of the covariance matrix using the recursive formula (9.1), (9.2) or b) updates sparse elements of the covariance using the covariance dwndate in (9.14). The decision is outlined in Algorithm 9.1. Note that this algorithm involves a simple *incremental* Gauss-Newton solver, but other nonlinear solvers (even batch solvers) are also suitable.

Algorithm 9.1: Covariance recovery algorithm selection in an NLS solver

1: **function** INCREMENTALGNWITHCOVS($\theta, \Sigma, \mathbf{v}, \mathbf{r}, \mathbf{z}_u, \Sigma_u, \text{max_iters}, \text{tol}$)

Require: Σ is the covariance matrix from the last step

Require: \mathbf{v} is a vector of ids of variable affected by the update

Require: \mathbf{z}_u is a new observation on the variables in \mathbf{v}

Require: Σ_u is covariance of the *observation* \mathbf{z}_u (not to be confused with Σ , the covariance of the *variables*)

2: $(\hat{\theta}, \hat{\mathbf{r}}) = \text{UPDATE}(\theta, \mathbf{v}, \mathbf{r}, \mathbf{z}_u, \Sigma_u)$

3: $(\hat{\Lambda}, \hat{\eta}, A_u) = \text{LINEARSYSTEM}(\hat{\theta}, \hat{\mathbf{r}})$

4: $\mathbf{o} = \text{AMD}(\hat{\Lambda})$

5: $\hat{\mathbf{R}} = \text{BLOCKCHOL}(\hat{\Lambda}, \mathbf{o})$ \triangleright Or use incremental factorization in Algorithm 7.4.

6: $\hat{\theta}_{\text{old}} = \hat{\theta}$

7: $\text{new_LP} = \text{false}$

8: $\text{GAUSSNEWTON}(\hat{\theta}, \hat{\mathbf{r}}, \Sigma, \hat{\mathbf{R}}, \hat{\mathbf{d}}, \hat{\Lambda}, \hat{\eta}, \text{new_LP}, \text{max_iters}, \text{tol})$ \triangleright Algorithm 7.2.

9: **if** new_LP **then**

10: $\hat{\mathbf{R}} = \text{BLOCKCHOL}(\hat{\Lambda}, \mathbf{o})$ \triangleright Need an up-to-date R factor.

11: **end if**

12: **if** $\hat{\theta}_{\text{old}} \neq \hat{\theta}$ **then** \triangleright See if the linearization point has changed.

13: $\hat{\Sigma} = \text{CALCULATECOVARIANCE}(\hat{\mathbf{R}}, \mathbf{o})$

14: **else**

15: $\hat{\Sigma} = \text{UPDATECOV}(\Sigma, \hat{\mathbf{R}}, \mathbf{o}, A_u, \mathbf{v})$

16: **end if**

17: **end function**

The two branches have different complexities. The first branch has complexity of $O(n_{\text{nz}}^2 n)$ in n_{nz} , the number of nonzeros of the $\hat{\mathbf{R}}$ factor and n , the sum of DOF of all the vertices [18]. The second branch is dominated by the complexity of $O(k n_{\text{nz}} n_u)$ where n_u is the sum of DOFs of vertices that are being updated. As a result, the second branch is much faster if only a few vertices are changing. In case that most of the vertices are being updated (e.g. after a linearization point change), the first branch becomes faster.

9.2.4.1 Sparse Blockwise Covariance Calculation

The proposed implementation of the recursive formula is slightly different from the other state of the art implementations iSAM [93] or g2o [106]. These use a hash map or a similar structure for fast lookup of the elements of the covariance matrix that were already calculated in the course of evaluating (9.1), (9.2). At the same time, they are query-driven: to calculate a specific value of the covariance matrix,

Algorithm 9.2: Blockwise covariance recovery.

```

1: function CALCULATECOVARIANCE( $\hat{\mathbf{R}}, \mathbf{o}$ )
Require:  $\hat{\mathbf{R}}$  is a block matrix with Cholesky of the current information matrix  $\hat{\Lambda}$ 
Require:  $\mathbf{o}$  is fill-reducing ordering used in the factorization of  $\hat{\mathbf{R}}$ 
2:    $\hat{\Sigma} = \text{NEWMATRIX}(\text{ROWS}(\hat{\mathbf{R}}), \text{COLS}(\hat{\mathbf{R}}))$   $\triangleright$  note that  $\hat{\Sigma}$  is a sparse block matrix
3:   for  $i = \text{SIZE}(\text{BLOCKROWS}(\hat{\mathbf{R}})) - 1$  down to 0 do       $\triangleright$  block rows in reverse
4:     for  $b = \text{SIZE}(\text{BLOCKROWS}(\hat{\mathbf{R}})[i]) - 1$  down to 0 do   $\triangleright$  blocks in reverse
5:        $j = \text{COLUMNOF}(\text{BLOCKROWS}(\hat{\mathbf{R}})[i][b])$   $\triangleright$  col. of  $b^{\text{th}}$  block in  $i^{\text{th}}$  row
6:        $\hat{\Sigma}_{j,i} = \mathbf{O}$   $\triangleright$  place a new zero block in  $\hat{\Sigma}$ , the size of  $\hat{\mathbf{R}}_{j,i}$ 
7:       for each  $\hat{\mathbf{R}}_{j,k}$  in  $\text{BLOCKROWS}(\hat{\mathbf{R}})[j]$  do       $\triangleright$  loop blocks forward
8:         if  $k > i$  then  $\triangleright$  read upper-triangular  $\hat{\Sigma}$  only
9:            $\hat{\Sigma}_{j,i} = \hat{\Sigma}_{j,i} + (\hat{\Sigma}_{i,k} \cdot \hat{\mathbf{R}}_{j,k})^{\text{T}}$ 
10:        else
11:           $\hat{\Sigma}_{j,i} = \hat{\Sigma}_{j,i} + \hat{\mathbf{R}}_{j,k}^{\text{T}} \cdot \hat{\Sigma}_{k,i}$ 
12:        end if
13:      end for
14:       $\hat{\Sigma}_{j,i} = \text{FINALIZECOVBLOCK}(\hat{\Sigma}_{j,i}, \hat{\mathbf{R}}_{j,j}, i, j)$ 
15:    end for
16:  end for
17:  return  $\text{PERMUTE}(\hat{\Sigma}, \mathbf{o}^{-1})$ 
18: end function

19: function FINALIZECOVBLOCK( $\Sigma, \mathbf{R}, i, j$ )  $\triangleright \Sigma$  and  $\mathbf{R}$  are dense blocks in  $\hat{\Sigma}$  and  $\hat{\mathbf{R}}$ .
20:   for  $k = \text{COLS}(\Sigma) - 1$  down to 0 do
21:     for  $l = (i \neq j ? \text{ROWS}(\Sigma) - 1 : k)$  down to 0 do  $\triangleright$  ternary operator
22:        $r_{\text{inv}} = 1/\mathbf{R}_{l,l}$ 
23:        $f = \Sigma_{l,k} + \Sigma_{l:\text{end},k} \cdot \mathbf{R}_{l:\text{end},k}$ 
24:       if  $i \neq j$  then
25:          $\Sigma_{l,k} = -r_{\text{inv}}f$   $\triangleright$  off-diagonal blocks of  $\hat{\Sigma}$ 
26:       else if  $k \neq l$  then
27:          $\Sigma_{k,l} = \Sigma_{l,k} = -r_{\text{inv}}f$   $\triangleright$  off-diagonal elements in diag. blocks of  $\hat{\Sigma}$ 
28:       else
29:          $\Sigma_{k,k} = r_{\text{inv}}(r_{\text{inv}} - f)$   $\triangleright$  elements on the diagonal of  $\hat{\Sigma}$ 
30:       end if
31:     end for
32:   end for
33:   return  $\Sigma$ 
34: end function

```

Algorithm 9.3: Incremental covariance update.

1: **function** UPDATECOV(Σ , $\hat{\mathbf{R}}$, \mathbf{o} , A_u , \mathbf{v})

Require: Σ is the covariance matrix to be updated

Require: $\hat{\mathbf{R}}$ is Cholesky of $\hat{\Lambda}$ with fill-reducing ordering

Require: \mathbf{o} is fill-reducing ordering used in $\hat{\mathbf{R}}$

Require: A_u is matrix of measurements since Σ

Require: \mathbf{v} is a vector of ids of variable affected by the update

2: $\mathbf{I}_v = \text{EYE}(\text{SIZE}(\hat{\mathbf{R}}))_{*,v}$ \triangleright select block columns of I corresponding to vertices \mathbf{v}

3: $\mathbf{I}_{v\mathbf{o}} = \text{PERMUTE}(\mathbf{I}_v, \mathbf{o})$

4: $\mathbf{T} = \hat{\mathbf{R}} \setminus \mathbf{I}_{v\mathbf{o}}$ \triangleright calculate block columns of $\hat{\Sigma}_v$, similar to (9.8)

5: $\mathbf{T} = \text{PERMUTE}(\mathbf{T}, \mathbf{o}^{-1})$

6: $\mathbf{Q} = \text{DENSE}(\mathbf{T}_{v,*})$ \triangleright select block rows, corresponding to \mathbf{v}

7: $\mathbf{M} = \text{DENSE}((A_u)_{*,v})$ \triangleright collect nonzero columns from A_u

8: $\mathbf{U} = \text{EYE}(\text{SIZE}(\mathbf{Q})) - \mathbf{M}\mathbf{Q}\mathbf{M}^\top$ \triangleright calculate dense \mathbf{U} , as in (9.10)

9: $\hat{\mathbf{B}} = \mathbf{T}_{0:\text{BLOCKROWS}(\Sigma),*} \mathbf{M}^\top$

10: $\hat{\Sigma} = \Sigma + \hat{\mathbf{B}}\mathbf{U}^{-1}\hat{\mathbf{B}}^\top$ \triangleright the update, as in (9.14)

11: $\text{ov} = \text{BLOCKCOLS}(\Sigma)$

12: $\text{nv} = \text{BLOCKCOLS}(\hat{\Sigma}) - \text{BLOCKCOLS}(\Sigma)$ \triangleright number of the new vertices

13: $\hat{\Sigma}_{*,\text{ov}:\text{end}} = \mathbf{T}_{*,\text{BLOCKCOLS}(\mathbf{T})-\text{nv}:\text{end}}$ \triangleright extend with cov. of the new vertices

14: **return** $\hat{\Sigma}$

15: **end function**

they start at that value and recursively work their way down the dependence tree, evaluating it in reverse order while backtracking.

In contrast, the proposed algorithm calculates the covariance matrix column by column, right to left, calculating only the queried covariance elements and all the elements at the same place as the nonzero elements of the $\hat{\mathbf{R}}$ factor. By the time the algorithm evaluates a specific element, it is guaranteed that all the references were already evaluated, eliminating the need for a hash map. A similar, but elementwise approach is described in [68]. Note that in Algorithm 9.2, the $\hat{\mathbf{R}}$ matrix is accessed by rows. The storage order is column-major though, so the implementation needs to transpose the structure of $\hat{\mathbf{R}}$ first and then it is possible to use this algorithm more efficiently. Note that the resulting covariance matrix is sparse: the algorithm does not calculate more elements, than [93, 106].

Once finished, the proposed algorithm permutes the calculated covariance matrix to the natural order, so that the block columns and block rows of $\hat{\Sigma}$ correspond to the variables of the optimized system. The covariance matrix is symmetric, and only the upper-triangular part is stored.

9.2.4.2 Covariance Update

Updating the covariance incrementally is significantly faster in the second branch of the [Algorithm 9.1](#). To calculate an update to the covariance matrix from the previous step, [Algorithm 9.3](#) closely follows the calculation outlined in [Section 9.2.2](#). Note that reordering the system matrix ($\hat{\Lambda}$ or $\hat{\mathbf{R}}$), e.g. as described in [Section 7.7.1](#) does not impede the incremental update and that the algorithm is valid, but not efficient when the linearization point changes.

The algorithm begins by evaluating \mathbf{T} , the block columns of $\hat{\Sigma}$ corresponding to the \mathbf{v} , the vertices which are being updated (lines 2 to 5). This is illustrated in [Figure 9.4](#), where \mathbf{T} comprises the highlighted columns of Σ on the left (or rows on the right, as Σ is symmetric). Note that the inverse fill-reducing ordering is applied so that the block rows of \mathbf{T} correspond to the variables of the optimized system.

To calculate the small matrix \mathbf{U} by directly following [\(9.10\)](#) would involve several sparse matrix products. In the proposed algorithm, dense calculations are used instead: the small portion of Σ used in the product is copied to a small dense matrix (\mathbf{Q} at line 6, corresponding to the highlighted blocks of Σ in the center of [Figure 9.4](#)). Similarly, nonzero columns of $\Lambda_{\mathbf{u}}$ are copied to another dense matrix (\mathbf{M} at line 7). The calculation of \mathbf{U} at line 8 is then performed using only small dense matrices, enabling better cache coherency and acceleration using [SIMD](#) instructions. This is equivalent to sparse evaluation of [\(9.10\)](#) and yields an identical result.

Finally, the additive update of Σ to $\hat{\Sigma}$ is calculated at line 10. Note that it is not needed to evaluate full dense $\hat{\mathbf{B}}\mathbf{U}^{-1}\hat{\mathbf{B}}^{\top}$. Instead, only the blocks of interest in Σ can be updated by using $\Delta\Sigma_{i,j} = \hat{\mathbf{B}}_i\mathbf{U}^{-1}\hat{\mathbf{B}}_j^{\top}$, in analogy to [\(9.7\)](#). In our implementation, this update is carried out in parallel. Some parts of $\hat{\Sigma}$ do not need to be updated, as they were already calculated using forward and backsubstitution (lines 2 to 5). These are the block columns, corresponding to the vertices being updated (\mathbf{v}). [Algorithm 9.3](#) uses this to *extend* $\hat{\Sigma}$ with covariances of the newly added vertices.

9.2.5 Experimental Evaluation

In this chapter, the focus was on testing the proposed algorithms on [SLAM](#) applications, but the applicability of the technique remains general. Many other applications from robotics such as active vision, planning in belief space etc. can benefit from the solutions proposed here.

The computational efficiency and precision of the method and its implementation were tested and compared with similar state of the art implementations, in particular, [iSAM \[95\]](#) and [g2o \[106\]](#). For [iSAM v1.7](#), revision 10 was used and for [g2o](#), [svn revision 54](#) was used. Both, [iSAM](#) and [g2o](#) use fairly similar implementation of the recursive formula [\(9.1\)](#), [\(9.2\)](#) together with a cache of already calculated

Table 9.1: Timing results (in seconds) of different state of the art covariance recovery implementations on multiple SLAM datasets, the best times is in bold.

Dataset	iSAM	g2o	SLAM ++	SLAM ++ total
Manhattan	206.58	180.42	4.37	13.88
10k	6712.03	5902.46	179.69	388.67
City10k	4585.15	3742.66	55.87	219.43
CityTrees10k	1009.91	938.97	30.98	60.41
Sphere	6051.73	5536.48	24.64	105.35
Intel	6.23	6.92	0.54	1.11
Killian Court	19.27	21.59	1.43	2.99
Victoria Park	310.57	293.09	13.89	37.11
Parking Garage	237.13	216.28	10.77	27.08

covariances, based on STL hash map containers. Although highly efficient, these implementations do not handle incremental updates of the covariance and instead recalculate it from scratch at every step. The proposed online covariance recovery is available in the SLAM ++ library³. Other implementations can easily benefit from the proposed scheme. The only requirement on the solver is to be able to solve for dense columns of Σ and to have explicit A_u or Ω .

The evaluation was performed on five simulated datasets; *Manhattan* [137], *10k* [71], *City10k* and *CityTrees10k* [94], *Sphere* [106] and on four real datasets; *Intel* [85], *Killian Court* [21], *Victoria Park* [133] and *Parking Garage* [106] (see Table 9.1). These are the datasets commonly used in evaluating NLS solutions to SLAM problems. The tests were performed on a computer with Intel Core i5 CPU 661 running at 3.33 GHz and 8 GB of RAM. This is a quad-core CPU without hyperthreading and with full SSE instruction set support. Each test was run ten times and the average time was calculated in order to avoid measurement errors, especially on smaller datasets.

9.2.5.1 Time evaluation

Table 9.1 shows the time performance of the incremental covariance recovery strategy in Algorithm 9.1 tested on the above-mentioned datasets and compared with g2o and iSAM implementations. The *block-diagonal* and the *last block-column* of the covariance matrix are recovered at every step in all the cases. These are the only elements of the covariance matrix required for taking active decisions based on

³ <http://sf.net/p/slam-plus-plus/>

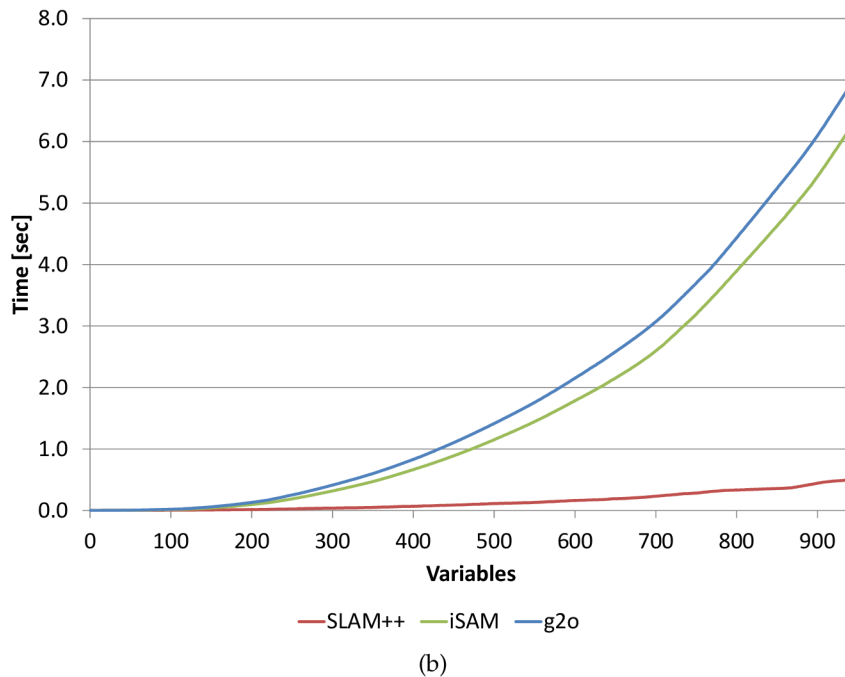
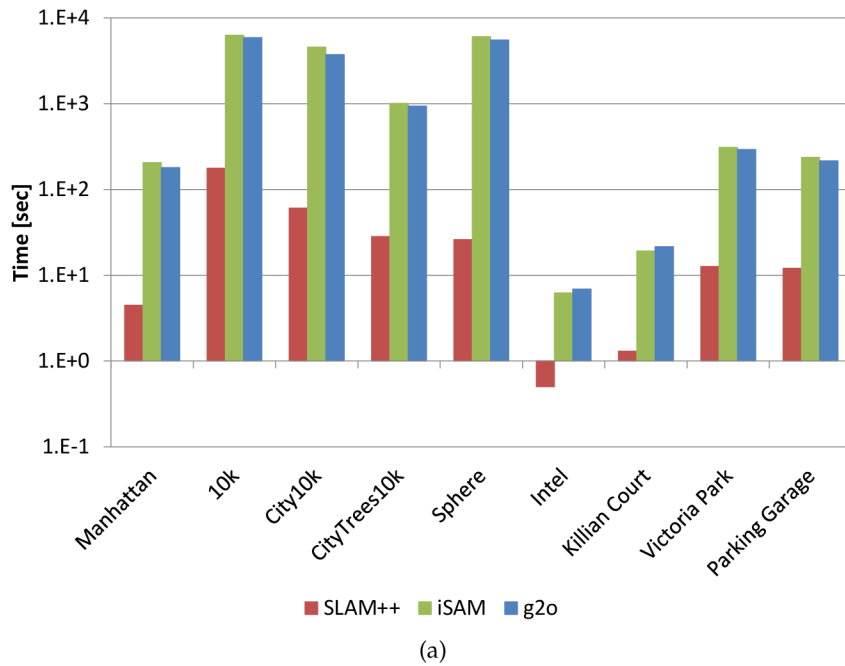


Figure 9.5: Covariance recovery performance evaluation; a) logarithmic plot of time on standard datasets and b) cumulative time on the *Intel* dataset.

the current estimation and efficient search for data association in an online [SLAM](#) application [89]. In incremental mode, the covariance is calculated after each variable added to the system (e.g. for the 10k dataset, it is calculated ten thousand times). The total time spent in solving the [SLAM](#) problem with covariance recovery is reported in the last column.

[Figure 9.5a](#) reports the covariance recovery time on logarithmic scale while [Figure 9.5b](#) shows the cumulative time of the incremental covariance computation on the *Intel* dataset during the execution of the algorithm. An approximate time com-

plexity was estimated from these readings using least squares. The time complexity for SLAM ++ $O(n^{1.77})$ is superior to the ones of g2o $O(n^{2.31})$ or iSAM $O(n^{2.36})$.

The performance of our incremental NLS solver in [PIŠ⁺13b] was also compared against GTSAM 2.3.1. However, the computation of the marginal covariances is not optimized for recovering all the block-diagonal elements in the current version of the GTSAM, therefore we excluded it from our comparisons. Nevertheless, we tested the available function for recovering the covariance of a single variable, the first variable (the most expensive one to calculate), against a similar function in SLAM ++, and this produced on *Manhattan*, 26.270 s GTSAM vs. 2.125 s SLAM ++, on *10k*, 261.880 s vs. 50.550 s, and on *Intel*, 1.429 s vs. 0.148 s.

In conclusion, the proposed implementation significantly outperforms all the existing implementations due to the proposed incremental covariance update algorithm and the blockwise implementation of the recursive formula.

9.2.5.2 Memory Usage Evaluation

Memory consumption of the above-mentioned implementations was also evaluated. The memory usage has been measured during two series of runs: with and without the marginal covariances computations. Figure 9.6a shows the overall memory usage from experiments performed on the *Intel* dataset and Figure 9.6b shows the memory allocation of marginal covariances calculation only.

The overall memory usage plot shows that SLAM ++ uses the least memory, which is achieved thanks to efficient implementation of matrix storage. The evaluation of the memory used by marginal covariances computation algorithm is comparable to g2o and iSAM. SLAM ++ performs pooled memory allocation, which can be seen as steps in the plot. This is advantageous, compared to the noisy allocation patterns of g2o and iSAM, which probably lead to more system calls and thus higher execution time.

9.2.5.3 Numerical Precision Evaluation

Since the proposed incremental update of the covariance is additive, it is likely that arithmetic errors in calculating the update will accumulate over consecutive steps, causing the solution to drift away from the correct values. Although no proof of numerical stability is offered here, we consider it is very important to show how the algorithm behaves in practice. A benchmark was performed on the *Intel* dataset, where the covariances were calculated using recursive formula, using the proposed method and using back and forward substitution to solve for a full inverse. The *Intel* dataset was chosen specifically because it contains just a handful of loop closures, causing the incremental covariance update to last for long periods, exceeding hundreds of steps.

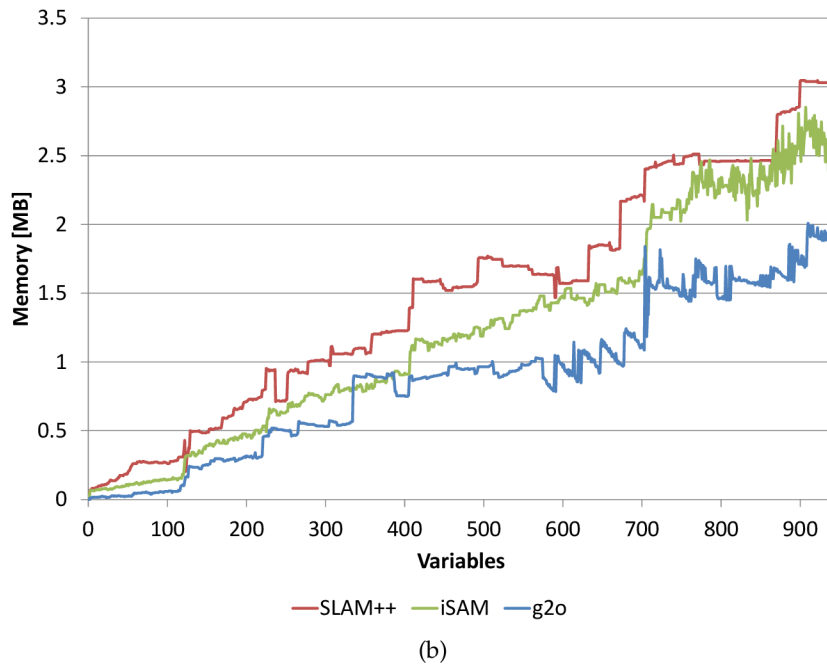
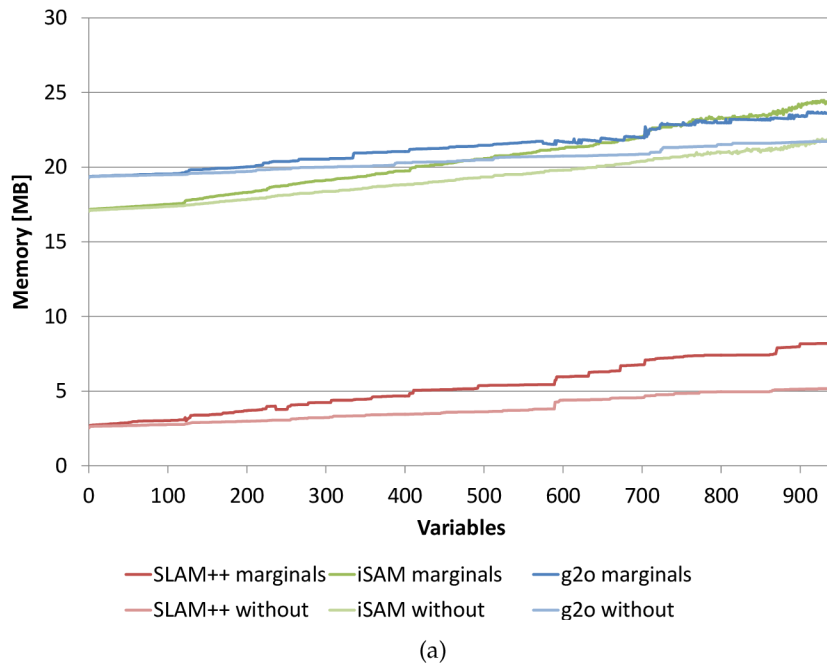


Figure 9.6: Memory usage benchmark; a) overall memory usage with and without marginal covariances computation, b) memory allocation of the covariances algorithm only.

Although being the slowest, backsubstitution was shown to be numerically *backward stable*. Therefore, the covariance calculated using back and forward substitution was used as a ground truth. The recursive formula in (9.1) and (9.2) is arguably less precise, as it reuses already calculated values of the covariance, potentially amplifying their error. The increment, $\Delta\Sigma$, is calculated from backsubstitution so it should be relatively precise, however the update is additive, allowing the error to slowly creep in. Figure 9.7 plots the relative norm of error of covariances, calcu-

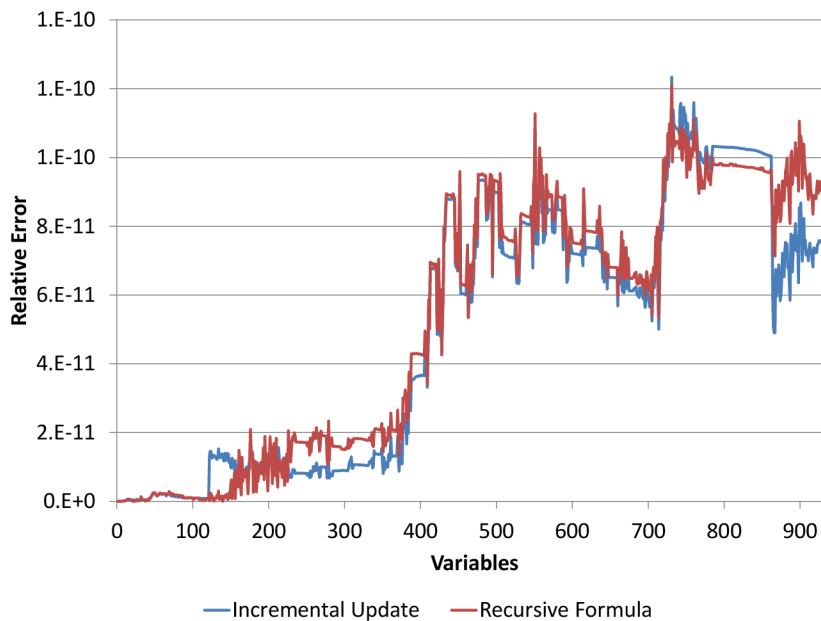


Figure 9.7: Covariance precision on the *Intel* dataset.

lated using the recursive formula and using the incremental update. It can be seen that the errors are quite correlated, incremental update having mostly lower error. This is given by the fact that the covariance for incremental update is initialized using the recursive formula after a linearization point change occurred. Generally, it shows that the covariance was calculated with error at the 10^{th} decimal place and that using the incremental update slightly increased precision, rather than decreasing it.

9.3 RECOVERING COVARIANCE IN SCHUR COMPLEMENTED SYSTEMS

In [Section 9.2](#), it was described how the covariances of the variables in an [NLS](#) estimation may be recovered efficiently and how the incremental updates to the system translate to the updates of the covariance matrix. However, as demonstrated in [Chapter 8](#), some of the problems can be solved more efficiently using Schur complement rather than by directly factorizing the system matrix Λ or Λ . In those cases, it is still possible to e.g. use the recursive formula [\(9.1\)](#) and [\(9.2\)](#) to obtain the covariances, but it comes at the cost of calculating an extra factorization of the entire system which would otherwise not be needed.

Thus, the goal is to solve $\Lambda \Sigma = I$ directly on the Schur complemented system:

$$\begin{pmatrix} A & U \\ U^T & D \end{pmatrix} \cdot \begin{pmatrix} \Sigma_p & \Sigma_{pl} \\ \Sigma_{pl}^T & \Sigma_l \end{pmatrix} = \begin{pmatrix} I_p & 0 \\ 0^T & I_l \end{pmatrix}. \quad (9.15)$$

where both Σ and the identity matrix I are partitioned the same way as Λ is partitioned in [\(8.2\)](#). Note that the subscripts here are only identifiers rather than element

indices. By taking Cholesky decomposition $S^T S \triangleq \text{Schur}(A)$, the covariances of the camera variables are:

$$S^T S \Sigma_p = I_p - UD^{-1}0^T = I_p \quad \text{so} \quad \Sigma_p = (S^T S)^{-1}, \quad (9.16)$$

and thus the recursive formula in (9.1) and (9.2) can be used efficiently.

The situation is more interesting in recovering the covariances of the landmarks. It would be possible to make use of $T^T T \triangleq \text{Schur}(D)$ and (9.16) to write:

$$\Sigma_l = (D - U^T A^{-1} U)^{-1} = (T^T T)^{-1}. \quad (9.17)$$

The matrix inverted here is positive definite and the recursive formula could be used again. However, the inverse A^{-1} is involved here: unless the underlying problem forms a bipartite graph which only really happens with vanilla forms of BA and as soon as e.g. intrinsic camera parameters, GPS or odometry measurements are introduced, A is no longer block diagonal and inverting it is much more difficult than inverting D in (8.2). Applying the Woodbury formula to (9.17) gives:

$$\Sigma_l = D^{-1} + D^{-1} U^T (A - UD^{-1} U^T)^{-1} UD^{-1}, \quad (9.18)$$

$$\Sigma_l = D^{-1} + D^{-1} U^T \Sigma_p UD^{-1}, \quad (9.19)$$

$$\Sigma_l = D^{-1} + D^{-1} U^T S^{-1} S^{-T} UD^{-1}. \quad (9.20)$$

Evaluating all of (9.20) would yield a dense matrix with the size approaching that of the full Σ which would be counterproductive. Instead, taking advantage of symmetry of Λ (and thus also of D and Σ), it is possible to write $B \triangleq S^{-T} UD^{-1}$ in order to get $\Sigma_{l,i,j} = D_{i,j}^{-1} + B_{i,*}^T \cdot B_{*,j}$ where D is blockwise representation of D . Note that UD^{-1} is a sparse matrix with the number of nonzero blocks in each column equal to the number of cameras that observe the point corresponding to that column; S^{-T} can be efficiently calculated using sparse sparse back-substitution.

Finally, to get the cross-covariances between the camera and the landmark variables, it is possible to use (8.3) with covariance in place of \mathbf{a} and identity on the right:

$$\Sigma_{pl} = (S^T S) \setminus (0 - UD^{-1} I_l), \quad (9.21)$$

$$\Sigma_{pl} = -\Sigma_p UD^{-1}. \quad (9.22)$$

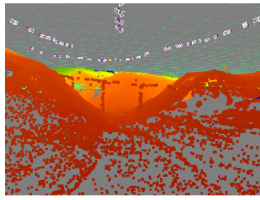
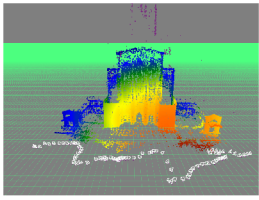
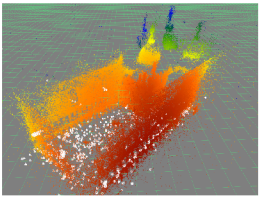
Again, computation can be saved by taking advantage of sparsity of the matrices so that recovering the full Σ_p is not necessary.

9.3.1 Experimental Evaluation

The proposed method for recovering marginal covariances of points was tested on two public datasets, the *Guildford Cathedral*⁴, Venice [106] and on the *Fast & Furi-*

⁴ can be obtained at <http://cvssp.org/impart/>

Table 9.2: Characteristics of the BA datasets used in covariance recovery evaluations.

	Fast & Furious 6	Guildford Cathedral	Venice
			
Cameras	160	92	871
Landmarks	136,453	57,957	530,304
Visibility	3.42 obs. / lm.	7.28 obs. / lm.	5.35 obs. / lm.
Λ	167.70 MB	142.93 MB	980.33 MB
Schur(A)	1.14 MB	1.04 MB	45.06 MB
S	1.73 MB	1.04 MB	84.60 MB

ous 6 dataset which was kindly provided by Double Negative Visual Effects⁵. Two additional methods were compared: recursive formula on Cholesky factor of the system matrix, and recursive formula on Schur(D) as in (9.17). More details about the datasets are listed in Table 9.2.

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. Memory consumption tests were performed on SGI UV2000 node, equipped with 14 of 8-core Xeon E5-4627 v2 at 3.3 GHz and 3.25 TB (Terabyte) of RAM; timing of these tests is denoted by the dagger[†] symbol. The turbo boost function of the Xeon CPUs was disabled for the benchmarks, so as to not make the results dependent on the variations in the temperature.

Sparse block schemes [PIS13a] were used throughout the whole implementation, which previously proved about an order of magnitude speedups for batch recursive formula [IPŠ⁺15]. Block matrix products and decompositions were accelerated by Tesla K20x GPU.

Times required to calculate the marginal covariances are reported in Table 9.3. The computation of the covariances of landmarks directly from Schur complement is the fastest for all tested datasets, followed by the use of recursive formula. The proposed method provides more than an order of magnitude speedup. The use of Schur(D) and recursive formula is prohibitive by both time and considerable memory requirements.

⁵ <http://www.dneg.com/>

Table 9.3: Timing results and the associated space requirements of the evaluated covariance recovery methods (best times in bold).

Dataset	Fast & Furious 6	Guildford Cathedral	Venice
Landmarks, using (9.20)	0.293 s	0.165 s	7.060 s
Size of sparse S^{-T}	2.97 MB	1.24 MB	109.79 MB
Chol(Λ)	0.951 s	1.251 s	16.856 s
Rec. formula all	3.493 s	3.308 s	82.689 s
Size of Chol(Λ)	93.33 MB	74.05 MB	572.52 MB
Schur(D)	149.999 s	160.662 s	4457.539 [†] s
Chol(Schur(D))	139 [†] hours	73 [†] hours	N/A
Rec. formula lm.	5 [†] hours	4459.647 [†] s	N/A
Size of Schur(D)	46.95 GB	37.87 GB	398.01 GB
Size of Chol(T)	493.43 GB	106.70 GB	~ 7.37 TB
Cameras, using (9.16)	0.045 s	0.028 s	38.809 s

The magnitudes of the calculated landmark covariances are displayed as false color, see Figure 9.2 or Table 9.2. From the colored view, it is apparent which parts of the reconstruction are more precise and which are not. The user can use this type of images to re-capture poorly reconstructed areas and obtain a high accuracy 3D reconstruction.

9.4 CHAPTER SUMMARY

In Section 9.2, a novel method for incrementally updating the covariance in NLS problems was introduced, which significantly speeds up the computation of the covariance matrices useful in a broad range of robotic applications. Problems which have a particular block structure were targeted, where the size of the blocks corresponds to the number of Degrees of Freedom of the variables. The advantage of the new scheme was demonstrated through an exhaustive comparison with the existing implementations on several available datasets. The tests show that the proposed scheme is not only about an order of magnitude faster, but also numerically stable. Error of the covariance calculated using the incremental update is, on average, lower than the error of the commonly used recursive formula.

In [IPŠS17], the usefulness of the incremental covariances calculation is demonstrated in the context of data association, where the number of expensive sensor registrations can be reduced by applying distance tests [89].

Methods for efficiently finding covariances in NLS problems which are solved using Schur complement, such as BA, were also proposed. The implementation of the formulas described in this chapter significantly outperformed the existing methods, by a factor of $20\times$ for *Guildford Cathedral*, $12\times$ for *Venice* and $12\times$ for *Fast & Furious 6*. At the same time, the memory consumption for calculating the inverse of square root of the Schur complement is comparable to the storage of the square root itself (which is required by the nonlinear solver), and is much smaller than the storage needed for square rooting the full system for recursive formula. Using the Schur complement of the landmarks is prohibitive as it requires tens to hundreds of GB of storage, not to mention its square root, which would require hundreds to thousands.

The calculated covariances can then be interactively displayed using false color rendering and used for quality assessment of the 3D reconstruction. The proposed methods are fast enough to be run on-set so that additional data capture can take place if the reconstruction quality is not good enough. For more details about the applications of efficiently recovering the covariances in digital cinema production, please refer to [BEK⁺15] or [PKP⁺15].

Part III

ACCELERATING THE CHOSEN ALGORITHMS ON GPU

This final part builds on the results and especially limitations of the previous one and proposes efficient GPU accelerated linear algebra solutions which can be used to accelerate the algorithms described so far.

ACCELERATING THE CHOSEN ALGORITHMS ON GPU

The previous parts proposed an efficient implementation of a Nonlinear Least Squares (NLS) solver library. It proved to outperform similar state of the art implementations, on high level due to algorithmic improvements and on low level due to sparse block matrix storage and operations design. Apart from Chapter 8 which employed simple GPU acceleration using existing libraries for *dense* operations, all the experiments were running on CPU only. However, several bottlenecks were identified:

SPARSE MATRIX MULTIPLICATION: used heavily in Schur complement, incremental Schur complement and covariance recovery.

SPARSE MATRIX TRANSPOSE: used in Schur complement, incremental Schur complement and covariance recovery. Although not directly a bottleneck, not having sparse transpose on GPU would necessitate copying the data back and forth, as well as CPU-GPU synchronization. The underlying operation is parallel sorting (the column-ordered entries of a sparse matrix are sorted by row, yielding a transpose matrix).

SPARSE MATRIX FACTORIZATION: general sparse factorization kernel will probably not yield a large speedup for SLAM applications as the matrices are very sparse [174]. On the other hand, a jagged diagonal or band-diagonal matrix factorization would be useful in Schur complement implementation and would reduce the memory and computation requirements compared to a full dense factorization.

BLOCK DIAGONAL MATRIX INVERSE: used in Schur complement, generally not a bottleneck (except if using AMICS ordering where the diagonal blocks can get large) but is easily parallelizable.

10.1 A BRIEF HISTORY OF GPU COMPUTING

The complexity of computer generated imagery has been steadily increasing for the past few decades, hand in hand with the plausibility of its results. From the first computer animated movies which took days and weeks to render on large mainframes to today's video games which admittedly look much more realistic and render at steady 60 frames per second on consumer hardware. On one hand, this

was made possible through the research and advances of algorithms and rendering methods. In the more recent years, special hardware for graphics computation acceleration appeared – the GPU.

This is a simplified and not entirely accurate view; chips for accelerating graphics far predate the term GPU and some of them could only draw spans rather than polygons, not to mention texturing.

At first, the GPUs could only draw z-buffered polygons with color and texture and much of the initial development was focused on increasing the raw numbers of vertices (or polygons) that could be sent to the rendering pipeline and on the number of pixels that could be filled each frame. While the GPUs were good at the latter task, working with vertex transforms and lighting computations was done in software rather than in hardware – especially because the applications used a variety of different tricks and approximations which would be hard to map to circuitry. It was also believed that a fast CPU would be able to keep the pace.

But nevertheless along came the hardware transform and lighting pipeline which could be configured for a few different kinds of lighting effects or e.g. fog computations, although performing e.g. skeletal character animation was difficult if not impossible. Still, an undisputed benefit is that it had freed the CPU for other tasks. This was later followed by register combiners which had more flexibility, and eventually by shaders which were short C-like programs.

Since the price to performance ratio of the GPUs sky-rocketed thanks to the digital entertainment industry and mass production, it is no surprise that they were popular also for non-graphics computations. Early applications of the non-programmable graphics pipeline included e.g. fast collision detection using z-buffering and stencil operations [127] or matrix multiplication using multi-texturing and blending functions [107]. Addition of programmable shaders allowed implementation of more complex algorithms, e.g. sparse matrix solvers [20].

The long tradition of abusing the graphics pipeline for purposes other than graphics was finally ended by the introduction of Application Programming Interfaces (APIs) for general purpose computations. Compute Device Unified Architecture (CUDA) was introduced by NVIDIA in 2008 and was intended as an extension of the C++ language for NVIDIA GPUs. In 2009, it was followed by a more general Open Compute Library (OpenCL) which targets many different kinds of parallel platforms, including GPUs. Both CUDA and OpenCL expose functionality hidden from the graphics APIs, such as random memory access (*scatter* in addition to *gather*), inter-thread communication using shared memory, atomics or double-precision instructions. This compelled most of the authors to abandon writing new implementations in shaders. Note that some of those features were later introduced to the graphics APIs in form of the *compute* shaders.

Rather than describing the inner workings of a GPU, such as the thread or memory hierarchy. Please, kindly refer to one of the GPU programming guides, e.g. [135], or other plentiful material available on this topic.

PARALLEL SORTING ON GPU

Efficient parallel sorting is an important building stone of many algorithms. Although parallel sorting algorithms have been researched extensively in the past, implementing the same algorithms on GPU presents a significant challenge, due to the necessary amount of communication and synchronization, not to mention high irregularity of memory accesses. In this chapter, a highly efficient implementation of radix sort is discussed. The ultimate goal is to support sparse linear algebra calculations, where sorting is often employed as a preprocessing step of matrix compression [38] in order to improve load balancing and to increase utilization of parallel processors [16] or e.g. in sparse matrix transpose. In Figure 11.1, there is a breakdown of the execution time of the current sparse matrix multiplication algorithms, running on GPU. In there, it is clearly visible that sorting takes a substantial portion of the time.

Sparse matrix multiplication is characteristic by *scattering* the elementwise products in not easily predicted pattern. In order to be efficient, it must calculate products in the order in which the matrices are stored (such as compressed sparse column [41]). When implemented in parallel, this scattering would cause a lot of conflicts where different threads would require access to the same element of the output matrix. To resolve this, the current implementations calculate the product as a set of destination coordinates and associated values, which are then sorted and compacted.

This puts the problem in a different perspective: the data to be sorted is produced by the GPU (e.g. by a matrix multiplication routine), and the sorted results are consumed by the GPU. Therefore, we are not burdened by having to transfer the data between CPU and GPU, much to the contrary: the conventional approach would be to only use GPU for large enough problems and to process small problems on the CPU. In our perspective, such processing would involve the prohibitive cost of data transfers and CPU-GPU synchronization. On the other hand, there is some prior knowledge about the range and distribution of the sorted data. The radix sort algorithm is able to use such knowledge to significantly accelerate the sorting, but still remains general.

When comparing the state of the art GPU accelerated libraries that provide sorting functionality, there is a significant performance gap: implementations based on CUDA achieve about twice the sorting rates of the OpenCL-based ones. The proposed implementation is intended to show that efficient sorting can be imple-

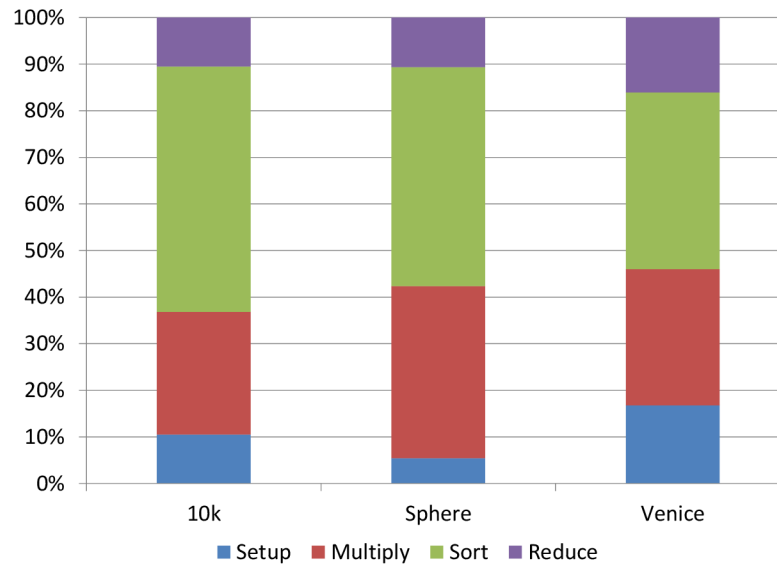


Figure 11.1: Relative amount of time spent in different phases of sparse matrix multiplication on GPU, measured on the CUSP library [128] and on matrices of standard SLAM (10k, Sphere) and BA (Venice) datasets.

mented even without advanced features exposed by CUDA, such as dynamic parallelism or thread voting. The proposed approach outperforms all of the compared implementations.

11.1 RELATED WORK

Some of the first attempts on efficient sorting on GPU [143, 102, 101] were implemented using the programmable shading pipeline, and were based on sorting network [13] approach. Govindaraju et al. [70] extended the idea to fully utilize the vector pipeline of the shading units and implemented a large-scale out-of-core sorting. The obvious disadvantage is a considerable overhead of using a graphics API, but general-purpose computing APIs did not exist yet. Sorting networks furthermore require relatively large number of passes, which grows with the size of the sorted sequence. These passes required communication through global (texture) memory, and the upper bound of performance was therefore relatively low.

One of the first influential sorting implementations in CUDA, Satish et al. [149] proposed to use the radix sort algorithm. Their method processed data in four passes that included local block sorting using 1-bit split operations [19], local histogram calculation, global prefix sum over histograms and finally reordering the data. Although this method is similar to the Algorithm 11.1 from this chapter, it is not optimal. The first local block sorting step was intended in order to improve memory access patterns in the last scattering step, which can have detrimental effects on performance if not properly handled. However, it is not work-efficient.

Sintorn and Assarsson [154] were able to develop a method based on a combination of merge sort and bucket sort. The bucket sort is used to improve parallelism at the later stages of sorting, where the number of lists to be merged becomes lower than the number of parallel processors. Their implementation, although based on comparison sorting algorithms, outperformed the work of Satish et al. [149] for arrays of 8 M elements, or more. One disadvantage of this method is the use of atomic counters to perform the bucket sort, and as such it depended on the distribution of the sorted data, as atomic operations on the same counter are subject to serialization in many parallel architecture, including GPUs.

The efficiency of radix sorting was improved by Ha et al. [76] by focusing on the arithmetic intensity of the sorting. To reduce the number of arithmetic operations in sorting, several optimizations such as the accumulation of three 10-bit histogram bins in a single 32-bit integer or the use of mixed-data structure are applied. It is based on the observation that bigger value types suffer less from irregular memory access patterns at the scatter phase. Therefore, array of key-value structures is preferred for this step, rather than the commonly used structure of arrays. As a result, about 30% greater sorting rate is achieved, compared to the Satish et al. [149] implementation.

Currently the fastest state of the art implementation is that of Merrill and Grimshaw [122] and [121], which was greatly influential also to the proposed method. They build on the work of Satish et al. [149] and also use the idea of accumulating four 8-bit histogram bins in a single 32-bit integer. Several novel ideas are introduced in these works, one of the most important ones being the reduction of the number of steps per radix to three, as in Algorithm 11.1 where lines 3, 4 and 5 – 7 can run each as one step that only requires global communication at the beginning or at the end. This reduction in global memory traffic increases the upper bound on sorting throughput. It is achieved by performing local sorting at the end of the scattering step, where it can be done in work-efficient manner.

11.2 PROPOSED IMPLEMENTATION

The next subsection contains a brief description and performance analysis of the radix sort algorithm, followed by a detailed description of our implementation and the methods used for optimizing it. The proposed algorithm consists of the same three steps as [121], but they are executed on GPU in just two steps, for reasons described below. Although the proposed implementation is written in OpenCL, the design considerations are with respect to NVIDIA hardware, and when referring to some particular hardware specifics, it is that of the NVIDIA platform, unless specified otherwise.

Algorithm 11.1: Segmented parallel radix sort.

```

1: function RADIXSORT(input)
2:   for each digitplace in {LSB, ..., MSB} do
3:     Calculate segment histogram of digits at digitplace
4:     Inplace global scan of all the histograms

5:     Segment scan of counts of digits at digitplace
6:     Add histogram scan to get global offsets
7:     Scatter temp ← input

8:     Swap input ↔ temp
9:   end for
10:  return input
11: end function

```

In the proposed algorithm design, the emphasis is on low arithmetic density, taking advantage of OpenCL just-in-time compilation model for flexible scheduling, and parallel programming with minimal synchronization using *warp-synchronous* programming where possible.

11.2.1 The Radix Sort Algorithm

Radix sort [103] is a stable sorting algorithm, suitable for sorting keys that map to integral values, such as integers or to certain extent the floating-point values. Note that this is converse to the widely used sorting paradigm that uses a comparison predicate and is implemented in e.g. C++ Standard Template Library. It works by grouping the given integer keys by their corresponding digits. This is done in successive fashion, starting with the least significant digits. Once grouped, the keys are then read out, starting with the group corresponding to the lowest value and maintaining the relative order of the keys in the same group. After going through all of the digits, the sequence is sorted. The parallel version of this algorithm, called split radix sort [19], relies on parallel prefix sum primitive extensively, to facilitate grouping of the sorted elements. Parallel prefix sum, or *scan*, can be implemented efficiently on GPU [150]. In order to extend radix sort algorithm to run efficiently on multi-processor machines such as GPUs, a notion of segments [19] is introduced. The sort can be broken down to local operations on the individual segments of the input sequence, which can be performed with reduced amount of communication between processors, working on different segments. The final sorting algorithm is described in Algorithm 11.1. A similar algorithm was used in [121].

Table 11.1: Memory complexities of Algorithm 11.1, the segmented radix sort.

Line of algorithm	Memory reads	Memory writes
3	n	$2^d m$
4	$2^d m$	$2^d m$
5 – 7	$n + 2^d m$	n

In the first step inside the loop, counts of digit values in each segment of the input sequence are calculated. Prefix sum of those counts gives the global position of the first occurrence of each digit in the output sequence, for each segment. Finally, the last step will calculate prefix sums of each digit, determining output position of each key in terms of the segment and by using the histogram prefix sum also the global output position. The output sequence of one loop iteration becomes input to the next one, output of the last iteration is the sorted sequence. In order to sort k -bit numbers, one needs to perform k/d iterations of the loop above, where d is size of a digit, in bits. Each segment histogram will therefore contain 2^d bins. An example of a single step of the loop is depicted on Figure 11.2.

Since sorting is certainly a bandwidth-limited operation, let us analyze the cost in terms of memory accesses. Given that the length of input sequence is n , and the hardware architecture dictates us to use m segments (where each segment corresponds to an individual parallel processor), the required bandwidth can be found in Table 11.1.

Since m is quite limited by the hardware (up to tens on GPUs, or hundreds on Intel MIC), and d is limited by register pressure, the memory complexity is roughly $3nk/d$. This can give us an idea about the upper bound of the sorting rates achievable on the current hardware. For example, NVIDIA GeForce GTX 780 has maximum bandwidth of 288.4 GB/sec, which can yield peak sorting rates up to 3.0 GKeys/sec for the common case of $k = 32$, $d = 4$. The proposed implementation is efficient, in the sense of achieving performance, comparable to this upper bound. Note that in the following text, the convention of binary units is used, where 1 M equals 1024^2 , 1 G equals 1024^3 , and so on.

11.2.2 Segmented histogram calculation

Histogram calculation is a fairly straightforward algorithm if implemented on a serial processor. On a parallel processor, two common approaches prevail. Sintorn and Assarsson [154] used atomic operations for incrementing the histogram bin counters, but despite recent architectural improvements, atomic operations still serialize if working on the same variable (the same histogram bin). The efficiency

Algorithm 11.2: Naïve histogram calculation.

```

1: function HISTOGRAM(input)
2:   histogram[16] = {0, 0, ..., 0}
3:   for each i in input do
4:     histogram[i] = histogram[i] + 1
5:   end for
6:   return histogram
7: end function

```

of histogram accumulation is then heavily dependent on the data, and is reduced up to $32\times$ on NVIDIA platforms in the worst case (since threads execute in groups of 32, called *warps*), or even slower on AMD platforms (similarly, threads execute in *waveforms* of 32 or 64 threads, depending on the specific GPU model).

The other solution, which the proposed implementation uses, is to trade time for space, having each thread accumulate in its private histogram, and have the threads reduce the histograms at the end. Segmented histogram is highly advantageous for GPU implementation, as there is no communication between the segments, and the reduction can take place entirely in the fast shared memory. The size of the segments is of a great importance, as it directly affects the performance. If the segments are too small, the costs of each thread initializing its private histogram with zeros and of the final reduction will easily outweigh the time, spent in the actual accumulation of the values, rendering the calculation inefficient. If, on the other hand, the segments are too large, there may not be enough segments to occupy all the streaming multiprocessors of the GPU. Many of the previous implementations restrict the size of the segment to a constant, for example implementation of Satish et al. [149] uses *tiles* of 1024 items. Instead, the proposed implementation, similarly to [121], uses variable length segments. The number of segments is chosen as a minimum that can keep the GPU fully utilized.

A distinguishing feature of the proposed algorithm is the choice of memory space for thread histogram storage. On GPU, there are several memory spaces with varying suitability. Global memory is mostly unsuitable for histogram accumulation, due to its latency. Shared memory is roughly two orders of magnitude faster, but it is accessed through a small amount of banks (16, or 32 on NVIDIA Fermi GPU and newer). If bank conflicts occur, the reads and writes are serialized. Therefore, even though not using atomic instructions, the accumulation would still be dependent on the data. Local memory [111] (not to be confused with local memory in OpenCL) is a memory space, specific to GPUs. It is a memory space, which is private to each thread. The values written to the local memory space are stored in L1 cache, but can be evicted to L2 and eventually to *global* memory (highly likely

Algorithm 11.3: Registered histogram accumulation.

```

1: function THREADPRIVATEHISTOGRAM(input)
2:    $(h_a, h_b, \dots, h_p) = (0, 0, \dots, 0)$ 
3:   for each  $(i, j)$  in input do ▷ process a pair of values at a time
4:      $\text{bin} = 1 \lll i$ 
5:      $\text{bin} = \text{bin} \cup (1 \lll j)$ 
6:      $\text{multiplicity}_{\log_2} = (i = j) ? 1 : 0$ 
7:      $h_a = h_a + ((\text{bin} \ggg 0) \cap 1) \lll \text{multiplicity}_{\log_2}$ 
8:      $h_b = h_b + ((\text{bin} \ggg 1) \cap 1) \lll \text{multiplicity}_{\log_2}$ 
9:      $\vdots$ 
10:     $h_p = h_p + ((\text{bin} \ggg 15) \cap 1) \lll \text{multiplicity}_{\log_2}$ 
11:  end for
12:  return  $(h_a, h_b, \dots, h_p)$ 
13: end function

```

for bandwidth-intensive applications). This memory space is used only for register spills and *addressable arrays*. This is due to the absence of register addressing. That means that code like in Algorithm 11.2 will actually store values in global memory (in the worst case – but it occurs with high probability), and will be dependent on the data.

Instead, the proposed histogram algorithm accumulates the histogram in registers. Due to the nature of the GPU execution model, to use branching to decide which histogram bin should be incremented would result in thread divergence, serialization and again dependence of execution speed on the data. On GPU, it is better to compare the data at the input to all histogram bins, and use the results of the comparison to increment all the histogram bins, for every item of the data. This approach, however, yields high arithmetic intensity and is only efficient if there are enough threads running to cover up the latency. Instead, bit masking operations are employed to calculate the comparison. That enables accumulation of several different values at once by simply or-ing their masks together. Special care needs to be taken for accumulating duplicate values. The final accumulation part is summarized in Algorithm 11.3.

Note that the \ggg and \lll operators represent bitwise shift to the left and to the right, respectively, while \cup and \cap represents logical *or* and logical *and*. Also, the algorithm accumulates two symbols at once, and for the sake of simplicity does not handle the situation of odd-sized input. The code can be further optimized by sacrificing several bits of accumulator precision, and instead of performing 2^b shifts of bin (16 in Algorithm 11.3), only one shift (by 0 and by 8 bits) is used and the (constant) binary masks are shifted instead. That reduces the work to 26

In vertex program specification, there is the ARL instruction, but its use is limited.

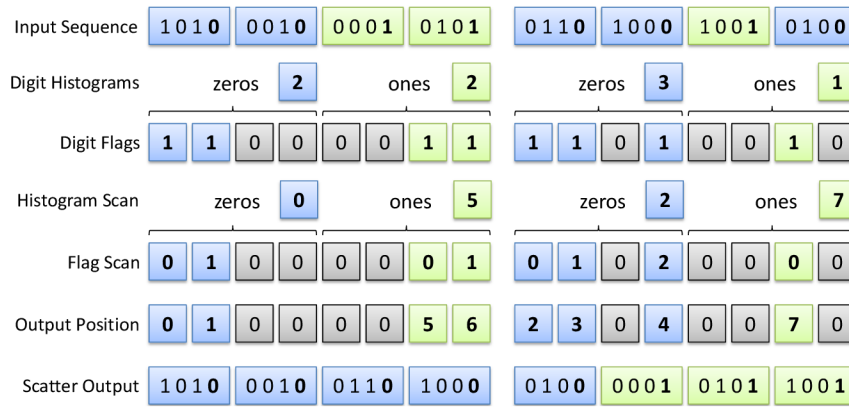


Figure 11.2: An example of segmented radix split operation for $b = 1$.

simple instructions per accumulated value. The accumulators need to be shifted at the end but that is a small, constant overhead. Note that the maximum size of the input is not reduced, thanks to parallelism.

After thread private histograms have been calculated, the values need to be reduced. The first part of the reduction is done in *warp-synchronous* manner, where each warp cooperatively reduces all its thread private histograms to a single histogram in shared memory. In order to completely avoid synchronization, each thread rotates its histogram bins by its id modulo 2^b . Afterwards, standard tree-based reduction is applied in shared memory. As a result, to reduce 512 histograms of 16 bins each, only four barrier synchronizations are required.

11.2.3 Fast Scan & Scatter

After accumulating the segment histograms, their prefix sum is calculated much like in [149], which will be used as a global destination offset for the sorted elements. Since the number of segments required to occupy the GPU is small, this step is not large enough to be efficiently issued as a separate kernel, and is fused with the last scattering step. Note that although this saves kernel execution, it does not save significant amount of communication and Table 11.1 still applies.

In order to perform scattering of the sorted sequence, global indices need to be calculated for each of the elements. Segmented prefix sum of histogram bin affiliation flags yields local ranks of the sorted elements. By adding value of histogram prefix sum for the corresponding segment and histogram bin, global position in the output sequence is obtained, as illustrated on Figure 11.2. This requires us to calculate 2^b prefix sums, each of the size of the segment, or alternately more shorter prefix sums with carry.

Several interesting observations can be made. The prefix sums are of binary flags, and sum up to segment length. This gives us knowledge of how many bits

are needed for the accumulators and it is possible to employ data-level parallelism. Ha et al. [76] perform accumulation of three 10-bit accumulators, similarly Merrill and Grimshaw [121] accumulate four 8-bit numbers in a single 32-bit variable.

It is possible to perform dynamic scheduling of accumulator precision in case of prior knowledge of the final sums, such as the histograms calculated in the first step of the algorithm. For example, a tile of 1024 element flags summed in 16 bins requires up to 128 bits, in 8 bins up to 64 bits. It is therefore possible to scan 1024×16 flags in two 64-bit numbers (always 8 in each), and if the distribution is favorable, a single 64-bit number suffices. This can be verified by solving the unordered partition problem for (1024, 16). This however relies on calculating segment histograms for relatively small segments which reduces performance on the current GPUs, and we choose to use Merrill's method. This technique is, however, relevant to the future GPUs that would have more multiprocessors or more registers.

In the proposed implementation, each thread calculates local scan of two flags. Warp-synchronous prefix sums with carry are used to calculate segment scan. Threads working on a single segment exchange sorted elements in shared memory as in [121] and write them out to the temporary array.

11.2.4 Register usage optimization

One of the disadvantages of register histogram accumulation described in Section 11.2.2 is the number of registers it uses (34 in our case). That directly affects possible number of workgroups, running on a single multiprocessor, and affects the capability to hide computational latency. In order to reduce the number of registers, a simple novel technique called *volatile stripping* is proposed. It is based on an observation that the OpenCL compiler allocates registers in a manner that will yield high processing speed, while the programmer has very little control over it. Declaring variables as *register* has no effect, and the compiler (NVIDIA 331.82) seems to ignore the `'-cl-nv-maxrregcount'` option.

In the histogram kernel, accumulation of the bins can be done in-place, but the compiler does not do that, possibly to improve pipelining. In our implementation, the histogram bin variables are declared as *volatile*. That makes the compiler generate code for storing the value of the variables in local memory. A post-processing step is applied to the generated assembly code, which uniquely identifies each variable based on its address in local memory, strips all the volatile load and store instructions, and instead assigns a single register where the variable is stored. Using this technique, we were able to reduce register use from 34 down to 27, significantly improving occupancy.

Since this technique is rather low-level, and while general, currently only implemented for NVIDIA PTX assembly format thus creating platform dependence, it was disabled in the performance evaluation in order to make fair comparison to the other OpenCL implementations, which are platform independent. Although volatile stripping possibly damages software pipelining, increased occupancy results in roughly 10% speedup for inputs of sufficient size to saturate the GPU memory subsystem.

Note that there is an initiative to make the OpenCL implementations use the same intermediate program representations derived from the one, used by the LLVM compiler. This initiative is called SPIR¹ and its most recent version SPIR-V is supposed to be adopted by OpenCL, OpenGL and Vulkan APIs. No tests using this intermediate representation were performed as the NVIDIA platforms do not seem to support it yet. Due to NVIDIA's commitment to support Vulkan, it is reasonable to expect that it will be supported in the future, however.

11.3 PERFORMANCE ANALYSIS

In this section we compare the timing results of radix sorting performed using the proposed implementation with similar state of the art implementations such as CUDPP 2.1, Thrust 1.6.0, CLOGS 1.2.0, CLpp v1 beta 3 and libCL 1.2.1. All of those libraries use the radix sort algorithm. Some of them also implement predicate-based sorting, but it is slower than radix sorting, and therefore of no interest in our application. The evaluation was performed by sorting vectors of random numbers of varying lengths (the same sequences were used for all the implementations). We also performed evaluation on sequences, produced by multiplying sparse matrices from The University of Florida Sparse Matrix Collection [39]. 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.

CUDA Data Parallel Primitives Library (CUDPP) is feature rich library with functions like parallel reduction, prefix sum, radix sorting, sparse matrix operations, random number generation and hashing. It supports comparison sorting, sorting optimized for strings, and radix sorting.

Back 40 computing (B40C) is another reusable parallel primitive library, developed in CUDA. It contains fast scalable radix sorting routines, designed around the allocation paradigm [123]. The B40C is now deprecated, the radix sorting code was reused in CUB and Thrust [82] libraries. We will focus on Thrust in our evaluation, as it is included in CUDA releases and is widely used. Thrust provides many func-

¹ See <https://www.khronos.org/registry/spir/>.

Table 11.2: Saturated GPU sorting performance in MKeys/sec.

Library	GPU Type			
	GTX 680		GTX 780	
	Key	Key-value	Key	Key-value
CUDPP	689.752	538.849	804.798	590.816
Thrust	696.706	540.675	792.496	621.417
CLOGS	451.049	276.837	503.756	366.238
CLpp	134.716	94.245	154.076	122.487
libCL	N/A	85.106	N/A	98.655
proposed	805.605	641.969	1119.422	892.055

tions, including predicate-based and radix sorting, with interface similar to the one of C++ Standard Template Library.

CLOGS is a mature OpenCL implementation, providing scan and sort primitives. Sorting of any combination of scalar or vector key and value type is supported, as well as sorting only keys. The implementation is “loosely based” on Merrill’s Back40Computing [121] radix sort implementation. CLOGS feature auto-tuning ability, which chooses the best parameters for target platform by exhaustively trying possible launch options, which are cached.

CLpp implements several sorting algorithms. Simple implementation of Radix Sort, as described by Blelloch [19], as described by Satish et al. [149], and a generic version due to the authors of the library. It offers functions for sorting keys or key-value pairs. The size of the value can be configured, the keys are expected to be 32-bit unsigned integers. The default sort implementation, which is used in the benchmarks is based on the paper of Satish.

libCL only offers limited sorting capability: it can only sort key-value pairs, and only up to $4M - 1$ of them. Also, both key and value must be 32-bit types, and the key is compared as 32-bit unsigned integer, reducing usability for sorting floating-point numbers. There is no support for sorted type specification.

It is apparent that the CUDA implementations are of better quality, and are influential to the mostly inferior OpenCL implementations. This is in part given by the supported hardware features: CUDA naturally supports advanced NVIDIA hardware functions, such as dynamic parallelism or warp voting functions, which are unavailable in OpenCL. These features are used in the CUDA implementations, giving them a certain advantage. The one disadvantage of CUDA is that it is compiled for certain hardware profiles, and when a new platform emerges, the binary must be updated. This is not the case with OpenCL, where the programs are compiled at

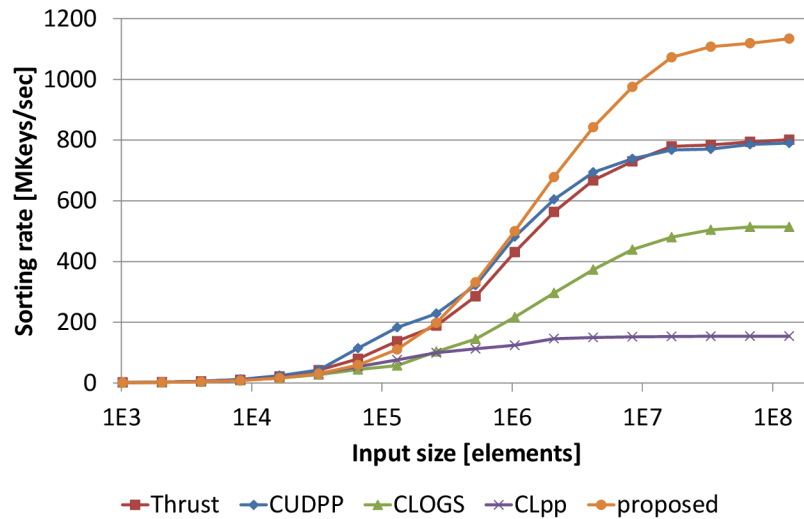


Figure 11.3: Sorting rates on 32-bit keys (higher is better).

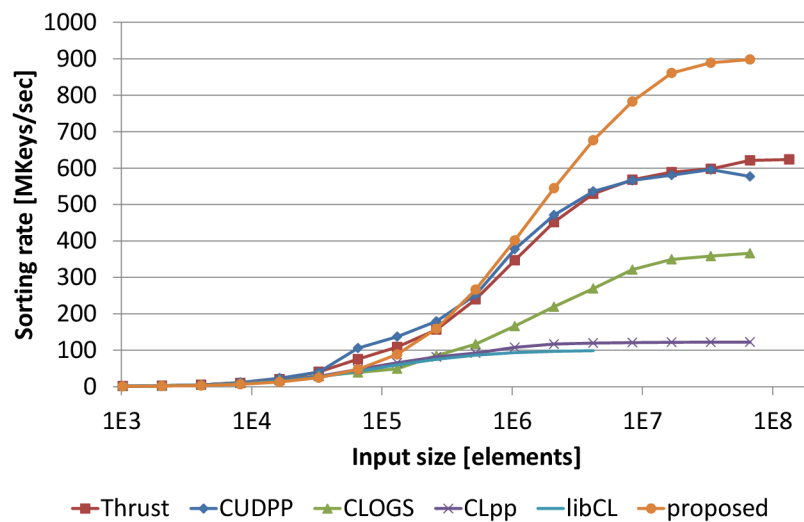


Figure 11.4: Sorting rates on 32-bit key-value pairs (higher is better).

run-time and can therefore adapt to new hardware immediately. This adaptation is only limited to number of registers, size of memory and similar device parameters.

All the tests were performed on a computer with NVIDIA GeForce GTX 680 and GTX 780, a pair of AMD Opteron 2360 SE CPUs running at 2.5 GHz and 16 GB of RAM. GPU drivers version 331.82 were used. CUDA implementations were linked against CUDA 5.5 SDK libraries. During the tests, the computer was not running any time-consuming processes in the background. Each test was run at least ten times 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 sequences. Explicit CPU-GPU synchronization was always performed, using `cuCtxSynchronize()` or `clFinish()`, respectively. Recorded times do not include any data transfers. The computer was running Windows 7 (64 bit) and all the tested libraries were compiled using Visual Studio 2008 SP1.

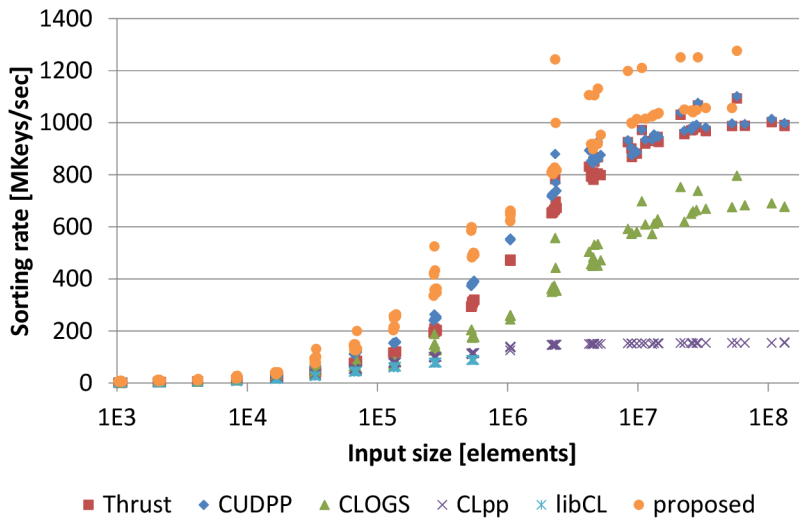


Figure 11.5: Sorting rates on 32-bit key-value pairs, keys were generated in sparse matrix multiplication (higher is better).

Summative results can be found in [Table 11.2](#). These were measured on random unsigned 32-bit numbers (care was taken so that the random numbers are not banded, but indeed span the whole 32 bits) and optionally 32-bit values. More detailed benchmarks are seen at [Figures 11.3](#) (keys only) and [11.4](#) (key-value pairs).

Since different implementations might react differently on the distribution of the sorted numbers, we also performed benchmarks by sorting element indices, obtained by performing sparse matrix multiplication, and recording destination row and column indices of results of every scalar product (see [\[38\]](#) for more details). Row and column indices are combined to a single key by multiplying column index by the number of rows and adding row index. Average runtime results on data generated by multiplying 160 of randomly chosen matrices from University of Florida Sparse Matrix Collection with their respective transposes are plotted in [Figure 11.5](#). Note that the proposed implementation consistently gains the fastest saturated sorting rates, only outperformed by CUDPP for very short sequences.

Also note that the authors of Thrust and CUDPP report greater sorting rates than measured, comparable with the proposed implementation. This is most likely due to the behavior on the particular [GPU](#) models, where our implementation is better optimized.

11.4 CHAPTER SUMMARY

In this chapter, a simple portable radix sort implementation suitable for [GPUs](#) was proposed. Although the achieved sorting rates are not much higher than the ones of the [CUDA](#) implementations, it improves over the fastest state of the art OpenCL implementations by nearly 50%. We achieved it by implementing fast histogram

accumulation in registers, using warp-synchronous synchronization-free operation. We proposed a novel technique of *volatile stripping*. Another proposed technique of dynamic allocation of accumulator precision is currently less efficient than state of the art, but will be applicable on bigger future GPUs.

This chapter presents a novel and highly efficient parallel algorithm for sparse matrix multiplication. Sparse matrix-matrix multiplication is an important algorithm, useful in a wide variety of scientific tasks, including among others computational chemistry and physics, graph contraction, breadth-first search from multiple vertices, algebraic multigrid methods, finite element methods or solving (non)linear systems using Schur complement [178].

The sparse matrix algorithms are usually tightly coupled to the sparse matrix storage formats they use. Two of the popular formats are compressed sparse column (CSC) [41] and compressed sparse row (CSR). Those are closely related; matrices stored in one are transposes of the matrices stored in the other. CSC stores matrices as a vector of prefix sums of numbers of nonzero elements in each column and two vectors storing element values and their respective rows. It is common for the elements in each column to be ordered by their row number. The use of the CSC format is assumed in the rest of this chapter, unless specified otherwise.

Let us recall that in matrix multiplication $C = A \cdot B$, each element of the product $C_{i,j}$ is a sum of products of the corresponding elements in the i^{th} row of A and the j^{th} column of B . The number of columns of A must match the number of rows of B . In CSC, it is straightforward to look up elements by column ($O(1)$) but not to look up elements by row ($O(n)$ in the number of nonzero elements), which would be needed to calculate the elements of C in ordered fashion (*gather*).

The original algorithm for sequential sparse matrix multiplication [75] is implemented e.g. in the popular CSparse package [41] (used by Google's Ceres solver and Street View), and is work-efficient in terms of its complexity being proportional to the number of Floating Point Operations (FLOPs). It is worth mentioning that this level of efficiency is only reached for the price of calculating a partially unordered representation of the product, which is still useful in practice, but it is not the canonical form.

The algorithm [75] is efficient by traversing the elements of B column by column (assuming the CSC storage is used; for CSR all the terms are transposed), where each element $B_{i,j}$ multiplies all the elements of A in the i^{th} column (the one corresponding to the *row* of the particular element $B_{i,j}$). Many of the other sparse matrix multiplication algorithms use this strategy. It produces partially ordered partial products (*scatter*), which need to be summed up. Gustavson [75] came up with an elegant way of quickly merging these partially ordered sequences.

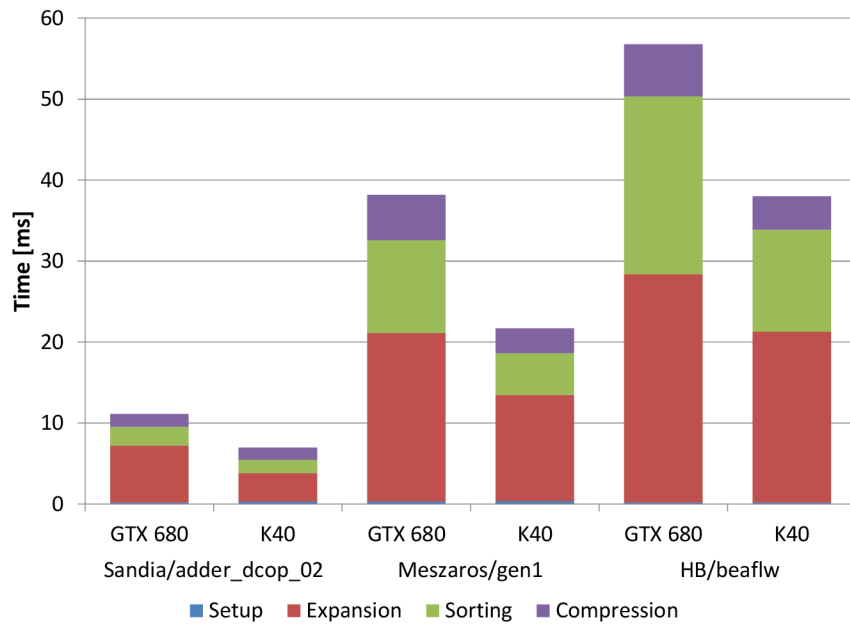


Figure 12.1: Time of different stages of the proposed algorithm.

Parallel sparse matrix multiplication algorithms (PSPGEMM in BLAS terminology), however, generally decompose the matrices to band or block submatrices and distribute the computation of the partial products to different processors. Similarly like in the previous case, the results need to be merged to form the final product, using sparse matrix addition in this case. This approach is further referred as a *coarse-grain* work subdivision, since the submatrices are typically relatively large. Packages [11, 59] use this approach.

12.1 RELATED WORK

Unlike dense matrix multiplication [2, 44, 34, 74, 60] which is very well researched and widely understood, sparse matrix multiplication [24, 22, 148] is much more challenging - and even more so in the hardware. Many papers titled “sparse matrix multiplication” actually refer to sparse matrix-dense matrix multiplication [120, 151], which is an extension of sparse matrix-vector multiplication (PSPGEMV or PCSRMM), an equally useful but nonetheless different algorithm.

The work of Buluç and Gilbert [24] discusses the challenges of designing and implementing scalable sparse matrix multiplication in distributed memory systems. Coarse-grain 1D decomposition of the work is considered, and two novel 2D algorithms are presented. The identified challenges are the load imbalance, the amount of work for partial result reduction and the communication overheads.

Matam et al. [120] explore several variations of the coarse-grain work division in a hybrid CPU-GPU algorithm: row-column, column-row and row-row. A heuristic is

proposed for the fastest row-row case that efficiently balances computational load between the CPU and the GPU. The load balancing is further extended for a special case of banded matrices. The work contains highly efficient implementations of both multiplication of two sparse matrices and a sparse with a dense matrix.

The work of Bell et al. [16] is strongly influential in the context of the later developments of GPU algorithms. It proposes the Expansion Sorting Compression (ESC) algorithm. The expansion stage is based on the producing a triplet form matrix of partial products, using the same operation ordering as used in [75]. To convert to CSC, the partial products need to be sorted and the entries contributing to the same element of the product need to be *reduced* (summed up) in the compression stage. The parallel primitives considered here are amenable to *fine-grain* work distribution. The method proposed in this chapter is based on the ESC algorithm.

Dalton et al. [38] further refined the work of Bell. Their implementation is public as the CUSP library [128] and it was used for comparisons with the algorithm proposed in this chapter. It focuses more intensively on the GPU platform-specific optimizations, such as avoiding passing data through global memory in favor of local memory and registers, especially in the sorting stage. The CSR storage is used, which is reflected in the three following paragraphs.

A permutation matrix is introduced, which orders the left operand by the work required to process a single row, facilitating load balancing. The product is later reordered, but the cost of doing so is reportedly relatively low.

The memory traffic of the expansion phase is further optimized for more regular coalesced accesses by casting the expansion process as a depth-first search on a layered bipartite graph of the nonzero elements of both factors.

The sorting phase is also optimized, by realizing that the produced expansion is partially ordered, only intra-row sorting is required. This is implemented as sorting many rows in the local memory at once. Very long rows which would not fit in the local memory are sorted using a global sort. Further reduction in sorting is achieved by a priori knowledge of the distribution of the bits of the sorted keys, and by copying lower bits of column indices to unused upper bits of row indices, effectively avoiding to have to sort simultaneously or sequentially by two keys.

Well known parallel algorithms, such as parallel sum (*reduction*), prefix sum (*scan*) [19, 150] or sorting [149, 121, 122] are used by the proposed technique. In this chapter, it is assumed that the reader is familiar with these operations.

12.2 ALGORITHM DESIGN

The algorithm introduced in this chapter is based on the ESC algorithm [16, 38]. However, the focus is on removing load imbalances and on simplicity, as especially

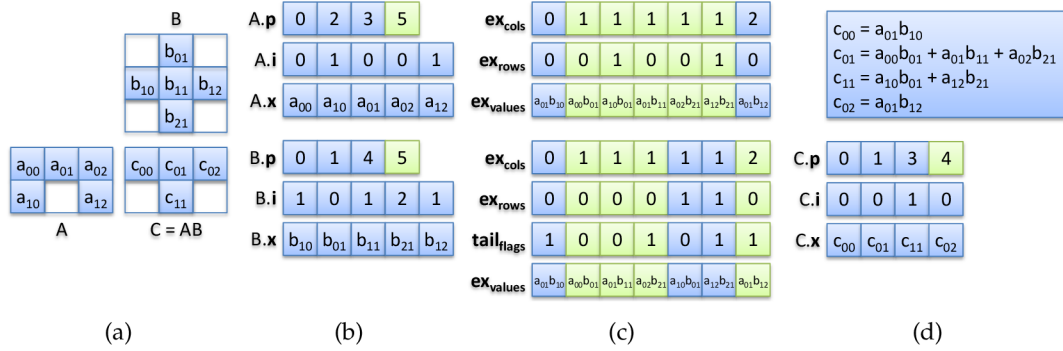


Figure 12.2: Data at the individual stages of the ESC algorithm¹, a) the factors and their product, b) CSC representation of the factors, c) top: expansion of the product, segments of product columns indicated by alternating color, bottom: sorted expansion, segments of product elements indicated by alternating color and d) values of the product and its final CSC form.

the improved ESC algorithm in [38] handles many special cases, depending on the memory space (local or global) and granularity (thread, warp or thread group) of each particular operation. In contrast, the proposed implementation only requires six custom kernels, some of which are merely a fusion of multiple general purpose operations such as scan, created for performance purposes only.

12.2.1 Expansion Stage

Although the first conceptual stage of the algorithm is expansion, on GPU it is not possible to directly proceed, without first knowing its size, as all the memory needs to be allocated before starting the computation. From [75], it is trivial to derive the exact size of the expansion:

$$\text{expansion}(A, B) = \sum_{j=1}^{\text{cols}(B)} \sum_{k=1}^{\text{nnzc}(B,j)} \text{nnzc}(A, \text{row}(B, j, k)), \quad (12.1)$$

where $\text{cols}(\cdot)$ gets the number of columns of a matrix, $\text{nnzc}(\cdot, \cdot)$ returns the number of nonzero elements in a specified column of a specified matrix and $\text{row}(\cdot, \cdot, \cdot)$ is the row of the given element in a column of a matrix. Note that all those are $O(1)$ array look-ups if the matrix is stored in CSC format. Also note that the expansion size is closely related to the number of FLOPs required to carry out the multiplication.

The expansion size dictates the memory cost of the ESC algorithm (the proposed variant as well as [16, 38]). Figure 12.3 plots a ratio of expansion size to the number of nonzeros in the product. In certain cases $100\times$ more storage than the final product is required (please, refer to Section 12.4 for the description of the dataset).

¹ An interactive demonstrator is available online at <http://www.fit.vutbr.cz/~ipolok/esc>.

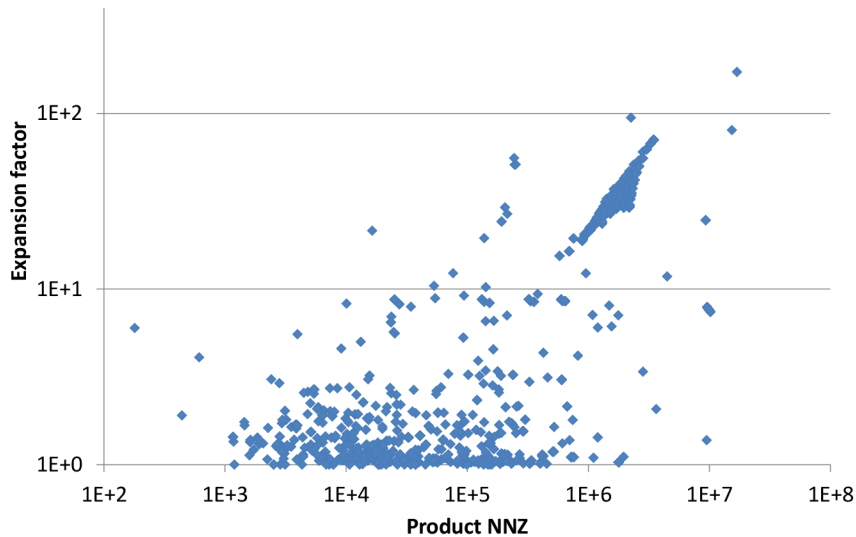


Figure 12.3: Expansion factor by the number of product nonzero entries.

Fortunately, it is possible to transparently subdivide the product by cutting the B matrix to several column slices, producing one slice of the product at a time.

The choice of granularity of expansion is crucial to load balancing. The proposed algorithm achieves perfect load balancing in the expansion stage by using the granularity of individual scalar products. To do that, it is necessary for each thread to find the elements of A and B to process. Here, the *interpolation search* [138] algorithm is employed. It is a special case of binary search where the pivot is chosen based on linear interpolation of the values of the endpoints of the searched interval with the needle as the argument. The average complexity is $O(\log \log n)$, worst case being $O(n)$. Interpolation search is not popular on CPU, as the linear interpolation is too expensive to outperform a regular binary search. However, it is a perfect fit for GPU where linear interpolation nicely hides under memory access latency and allows to find the needle in fewer steps and with much less branching.

The expanded scalar products are essentially the product matrix in the C00 format; they can be stored in three vectors of the same length, $\mathbf{ex}_{\text{cols}}$ contains column indices, $\mathbf{ex}_{\text{rows}}$ contains row indices and $\mathbf{ex}_{\text{values}}$ contains values of the elements (see Figure 12.2b). Note that the sparse multiplication algorithm generates a partially ordered expansion, where $\mathbf{ex}_{\text{cols}}$ is ordered and $\mathbf{ex}_{\text{rows}}$ consists of many short ordered runs (given by the rows of elements in the columns of A, which are typically ordered).

For comparison, the complexity of binary search is $O(\log n)$ in both average and worst cases.

12.2.2 Sorting Stage

The approach in [16] is to use a single global sort. On GPU, the most efficient sort implementations use radix sort [121] with complexity $O(kN)$ where k is pro-

portional to the number of bits of the key. In the case of keys generated by the expansion, the number of bits is given by base 2 logarithm of the number of rows and columns, respectively, and this knowledge can be used to accelerate the sort.

The radix sort may, however, not be the most efficient for a sequence which is already nearly sorted. As the sort starts, the expansion will be first ordered by the least significant bits of the keys, corresponding to the row indices. This will shuffle the column indices which were already ordered at the beginning. The elements are moved by long distances, leading to large amount of potentially uncoalesced global memory traffic. These will be ordered again in the later stages of the sort, by the most significant bits of the keys which correspond to the column indices, leading to long distance movement again.

The radix sort is efficient on GPU if the sorted elements are only reordered by small distances, as such reordering can be performed in the local memory. This is achieved by sorting it in segments corresponding to the individual columns of the product (Figure 12.2b top), instead of sorting the whole expansion at once. The individual segments can be sorted in parallel. Now the elements are only reordered by relatively short distances, leading to better write coalescing and leaving ample opportunity to do the sorting in the local memory. However, load balancing issues arise, as the lengths of expansions of the individual columns can vary wildly [38].

In the context of GPU computing, some operations have *segmented* variants, e.g. a *segmented scan*. Its input is a vector of values to calculate the scan of, and a vector of *head flags*, a binary vector with ones at the positions of segment starts. Note that segmented operation is performed on the bulk of data rather than on the individual segments, and thus requires no explicit load balancing. Unfortunately, radix sort is not a good candidate for segmented implementation, as it would lead to both runtime and space tolls: the key bit histograms would need to be evaluated per each segment and the reordering would also need to take place per segment, leading to more load balancing issues. Fortunately, for merge sort, segmented variants exist², and the performance toll, compared to the non-segmented variant, is negligible. By using segmented sort, the time of the sorting stage was significantly reduced; one can compare Figure 12.1 where sorting takes only 34%, with Figure 4 in [38] where it is closer to 63% of the total runtime.

12.2.3 Compression Stage

Once the expansion is sorted, the compression is a simple task of calculating sums of elements with the same row and column, which are now in contiguous segments of the expansion (see Figure 12.2b bottom). A simple segmented reduction can be

² One such implementation can be found at <http://nvlabs.github.io/moderngpu/segSORT.html>.

Algorithm 12.1: Setup stage of PSpGEMM.

```

1: function GEMM(A, B)
2:    $\mathbf{b}_{\text{cols}} = \text{ALLOCINT}(\text{NNZ}(B))$ 
3:    $\mathbf{b}_{\text{prods}} = \text{ALLOCINT}(\text{NNZ}(B) + 1)$ 
4:   kernel (i = 0 ... NNZ(B))
5:      $\mathbf{b}_{\text{cols}}[i] = 0$ 
6:     row = B.i[i]
7:      $\mathbf{b}_{\text{prods}}[i] = A.\mathbf{p}[\text{row} + 1] - A.\mathbf{p}[\text{row}]$ 
8:   end kernel ▷ the last element of  $\mathbf{b}_{\text{prods}}$  not initialized
9:   kernel (i = 0 ... COLS(B))
10:     $\mathbf{b}_{\text{cols}}[B.\mathbf{p}[i + 1] - 1] = 1$ 
11:  end kernel
12:   $\mathbf{b}_{\text{cols}} = \text{EXCLUSIVE SCAN}(\mathbf{b}_{\text{cols}})$ 
13:   $\mathbf{b}_{\text{prods}} = \text{EXCLUSIVE SCAN}(\mathbf{b}_{\text{prods}})$ 
14:  exp_size =  $\mathbf{b}_{\text{prods}}[\text{NNZ}(B)]$  ▷ expansion size

```

used to calculate the sums, while the head flags can be calculated as a difference of row and column numbers between consecutive expansion elements. Note that similarly to expansion stage, the size of the compressed form needs to be calculated first (e.g. as a sum (reduction) of the head flags) so that the memory to store the results can be allocated, unless the size of the product is known beforehand.

When handling matrices with large elements, such as long double or especially block matrix elements (i.e. dense blocks), it is beneficial to reorder the operations slightly: instead of storing the *values* of the partial products in the expansion, store only the *pointers* to the operands and calculate the products themselves during compression. This reduces both the size of the expansion and the memory traffic of sorting it.

12.3 IMPLEMENTATION

The proposed algorithm was implemented in OpenCL, and is presented in Algorithms 12.1, 12.2 and 12.3. This separation to parts is given by the need to allocate memory, which requires CPU intervention. The algorithm therefore requires CPU-GPU synchronization twice, at the beginning and after the end of Algorithm 12.2. This may be omitted if the allocation sizes or their upper bounds are known beforehand. Note that all the allocated buffers reside in the GPU memory.

In the algorithms, several conventions are followed. For any matrix M stored in the CSC format, $M.\mathbf{p}$ is the prefix sum of nonzeros in each of its columns, $M.\mathbf{i}$ is the vector of row indices of nonzero elements and $M.\mathbf{x}$ is the corresponding vector

Algorithm 12.2: Expansion and sorting stages of PSpGEMM.

```

15:    $\mathbf{ex}_{\text{cols}} = \text{ALLOCINT}(\text{exp\_size})$ 
16:    $\mathbf{ex}_{\text{rows}} = \text{ALLOCINT}(\text{exp\_size})$ 
17:    $\mathbf{ex}_{\text{values}} = \text{ALLOCFLOAT}(\text{exp\_size})$ 
18:    $\mathbf{ex}_{\text{hf}} = \text{ALLOCBIT}(\text{exp\_size})$  ▷ head flags bit array
19:   kernel ( $i = 0 \dots (n = \text{GPU}_{\text{hardware threads}})$ )
20:      $\text{begin} = \lfloor \text{exp\_size} \cdot i / n \rfloor$ 
21:      $\text{count} = \lfloor \text{exp\_size} \cdot (i + 1) / n \rfloor - \text{begin}$ 
22:      $\text{elemB} = \text{UPPER\_BOUND}(\mathbf{b}_{\text{prods}}, \text{begin}) - 1$ 
23:      $\text{col\_skip} = \text{begin} - \mathbf{b}_{\text{prods}}[\text{elemB}]$ 
24:     for ( $\text{prod} = 0; \text{prod} < \text{count}; ++ \text{elemB}$ ) do
25:        $\text{rowB} = \mathbf{B}.i[\text{elemB}]$ 
26:        $\text{elemA} = \text{col\_skip} + \mathbf{A}.p[\text{rowB}]$ 
27:        $\text{endA} = \mathbf{A}.p[\text{rowB} + 1]$ 
28:       while ( $\text{elemA} < \text{endA}$  and  $\text{prod} < \text{count}$ ) do
29:          $\text{dest} = \text{begin} + \text{prod}$ 
30:          $\text{cur\_col} = \mathbf{ex}_{\text{cols}}[\text{dest}] = \mathbf{b}_{\text{cols}}[\text{elemB}]$ 
31:          $\mathbf{ex}_{\text{rows}}[\text{dest}] = \mathbf{A}.i[\text{elemA}]$ 
32:          $\mathbf{ex}_{\text{values}}[\text{dest}] = \mathbf{A}.x[\text{elemA}] \cdot \mathbf{B}.x[\text{elemB}]$ 
33:          $\mathbf{ex}_{\text{hf}}[\text{dest}] = \text{cur\_col} > \mathbf{b}_{\text{cols}}[\text{elemB} - 1]$ 
34:          $++ \text{elemA}, ++ \text{prod}$ 
35:       end while
36:        $\text{col\_skip} = 0$  ▷ skip in the first iteration only
37:     end for
38:   end kernel

39:    $\text{SEGMENTEDSORT}(\mathbf{ex}_{\text{hf}}, \mathbf{ex}_{\text{rows}}, \mathbf{ex}_{\text{values}})$ 
40:    $\text{tail\_blocks} = \lceil \text{exp\_size} / \text{block\_size} \rceil$ 
41:    $\mathbf{tail\_counts} = \text{ALLOCINT}(\text{tail\_blocks} + 1)$  ▷ or reuse  $\mathbf{b}_{\text{prods}}$  (unused below)
42:   kernel ( $i = 0 \dots \text{exp\_size} - 1$ )
43:     local int  $\mathbf{flags}[\text{block\_size}]$  ▷ in local memory
44:      $\mathbf{flags}[i] = \mathbf{ex}_{\text{cols}}[i] < \mathbf{ex}_{\text{cols}}[i + 1]$  or  $\mathbf{ex}_{\text{rows}}[i] < \mathbf{ex}_{\text{rows}}[i + 1]$ 
45:      $g = \lfloor i / \text{block\_size} \rfloor$  ▷ cooperating thread group
46:      $\mathbf{tail\_counts}[g] = \text{COOPERATIVE\_REDUCE}(\mathbf{flags})$ 
47:   end kernel
48:    $\mathbf{tail\_counts} = \text{EXCLUSIVE\_SCAN}(\mathbf{tail\_counts})$ 
49:    $\text{productNNZ} = \mathbf{tail\_counts}[\text{tail\_blocks}] + 1$ 

```

Algorithm 12.3: Compression stage of PSpGEMM.

```

50:   C.p = ALLOCINT(COLS(B) + 1)
51:   C.i = ALLOCINT(productNNZ)
52:   C.x = ALLOCFLOAT(productNNZ)
53:   kernel (i = 0 ... exp_size - 1)
54:     g =  $\lfloor i/\text{block\_size} \rfloor$  ▷ cooperating thread group
55:     col_tail =  $\text{ex}_{\text{cols}}[i] < \text{ex}_{\text{cols}}[i + 1]$ 
56:     elem_tail =  $\text{ex}_{\text{rows}}[i] < \text{ex}_{\text{rows}}[i + 1]$  or col_tail
57:     local int flags[block_size] ▷ in local memory
58:     flags[i] = elem_tail
59:     flags = COOPERATIVE_SCAN(flags)
60:     compressed_index = tail_counts[g] + flags[i]
61:     if (elem_tail and i < exp_size) then
62:       C.i[compressed_index] = i ▷ write indices of
63:     end if ▷ reduced values of elements in expansion
64:     if (col_tail and i < exp_size - 1) then
65:       C.p[ $\text{ex}_{\text{cols}}[i] + 1$ ] = compressed_index + 1
66:     end if ▷ write positions of beginnings of columns
67:   end kernel
68:   C.p[0] = 0 ▷ need to write this explicitly
69:    $\text{ex}_{\text{values}}$  = SEGMENTEDREDUCTION(C.i,  $\text{ex}_{\text{values}}$ )
70:   kernel (i = 0 ... productNNZ)
71:     expansion_index = C.i[i]
72:     C.i[i] =  $\text{ex}_{\text{rows}}$ [expansion_index]
73:     C.x[i] =  $\text{ex}_{\text{values}}$ [expansion_index]
74:   end kernel
75:   return C
76: end function

```

of values of the elements. The parallel GPU kernel calls are denoted by **kernel**, and the (one-dimensional) execution domain is specified as $i = 0 \dots n$, where i is the name of the variable holding the thread id, and n is the required number of threads (thread with id $n - 1$ is the last thread).

In the setup stage (Algorithm 12.1), the \mathbf{b}_{cols} vector is filled with column indices of each corresponding element of B , making B available in both C00 (intermediate) and CSC (input) formats. This allows $O(1)$ lookup of column of any element of B in the later stages of the algorithm. Additionally, each element of $\mathbf{b}_{\text{prods}}$ contains the amount of work required to multiply all the *preceding* elements of B . This will be further used to facilitate load balancing at the expansion stage. The last element

contains the total amount of work, which equals the expansion size. Note that the kernel at line 9 needs to be modified if B is known to be rank deficient (then the number of succeeding empty columns needs to be added to each 1 in $\mathbf{b}_{\text{prods}}$, and care must be taken to not write to index -1). These changes were omitted in order to save space.

The expansion stage (Algorithm 12.2) begins by allocation of the arrays to hold the expanded values. The expansion is performed by the number of threads necessary to saturate the GPU (denoted $\text{GPU}_{\text{hardware threads}}$), or less if the expansion is smaller than that. Each thread will calculate the same number of scalar products, as discussed in Section 12.2.1. A range of scalar products to carry out (begin, count) is allocated for each thread, which then looks up $\mathbf{b}_{\text{prods}}$ for the element of B where to start multiplying (line 22). UPPER_BOUND is a standard binary search function: for an ordered vector and a value, it returns the right-most position where this value could be inserted without violating the ordering.

The inner loop at line 28 iterates over elements of a particular column of the A matrix, while the outer loop (line 24) takes care of advancing onto the next columns. Note that col_skip is used to start the loop in the middle of a column, should that be required to equally balance the workloads. Also note that if A is known to be rank deficient, the outer loop may need to advance multiple times, until reaching a non-empty column (such that $\text{elemA} < \text{endA}$ before entering the inner loop).

Once the expansion is calculated, the ex_{rows} , $\text{ex}_{\text{values}}$ pairs can be sorted while using the head flags as segment markers (note that the beginning of the first segment is implied and does not need to be explicitly represented). Finally, once the expansion is sorted, the boundaries of the elements and the columns can be easily spotted, and the number of nonzeros of the final product can be calculated, using the kernel at line 42. The variable block_size refers to the size of the blocks of the EXCLUSIVESCAN kernel, which is selected at runtime to best fit the target GPU.

In the final compression stage (Algorithm 12.3), the storage for the product is calculated. In the first kernel of this phase, the expansion is scanned for column tails (changes in ex_{cols} , line 55) and element tails (changes also in ex_{rows} , line 56). The scan of the element flags gives the element index in the compressed matrix. Note that the reduction of these flags was already calculated in the previous stage (line 46), which could be promoted to a scan to avoid recalculation, but storing the scans would require $O(\text{expansion size})$ memory and would be disadvantageous from both memory requirements and computational time standpoints.

Once the global index in the compressed matrix is known, indices of the final values of the elements in $\text{ex}_{\text{values}}$ can be written (C.i can be reused as temporary storage), and C.p can be filled. Again, if the product is rank deficient, care needs to be taken: C.p would need to contain runs of multiple occurrences of the same

index (including the zero index at the beginning), corresponding to the runs of empty columns.

Finally, the expansion values are summed up using segmented reduction, with `C.i` serving as tail flags, leaving the final values of the elements of the product at the tail positions in `ex_values` (line 69). The last kernel (line 70) merely copies these values to their compressed destinations in `C.p` and rewrites `C.i` by the corresponding row indices. Note that this kernel could be fused with the segmented reduction.

12.4 RESULTS

In this section, the timing results of sparse matrix multiplication performed using the proposed implementation³ are compared with a similar state of the art implementation, CUSP 0.3.1 [128]. It was also compared to CSparse 1.2.0 [41], which runs on the CPU⁴. Despite all effort, we were unable to find any existing OpenCL PSpGEMM implementations. The evaluation was performed by all-to-all multiplication of sparse matrices from The University of Florida Sparse Matrix Collection [39] and their transposes (for matrices which share a common dimension). 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.

All the tests were performed on a computer with NVIDIA GeForce GTX 680 (3 GB RAM) and Tesla K40 (12 GB RAM), a pair of AMD Opteron 2360 SE CPUs running at 2.5 GHz and 16 GB of RAM. In both cases, the program was compiled as x64, and both CUDA and OpenCL used 64-bit pointers. GPU drivers version 344.48 were used. CUDA implementations were linked against CUDA 6.5 SDK libraries. During the tests, the computer was not running any time-consuming processes in the background. Each test was run at least ten times 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. Explicit CPU-GPU synchronization was always performed, using `cuCtxSynchronize()` or `clFinish()`, respectively. ECC was disabled on the Tesla GPU.

Our implementation works with the CSC format. The implementations working with CSR format had their matrices converted (transposed) accordingly. Recorded times do not include the conversion or data transfers. The benchmarked version of the proposed algorithm handles all the rank deficient cases described in Section 12.3 in a fully general way, without requiring prior detection or specialized

³ The implementation of the proposed algorithm is available, at <http://sf.net/p/blockmatrix/>.

⁴ CSparse is used as an orientative example, more efficient CPU implementations exist.

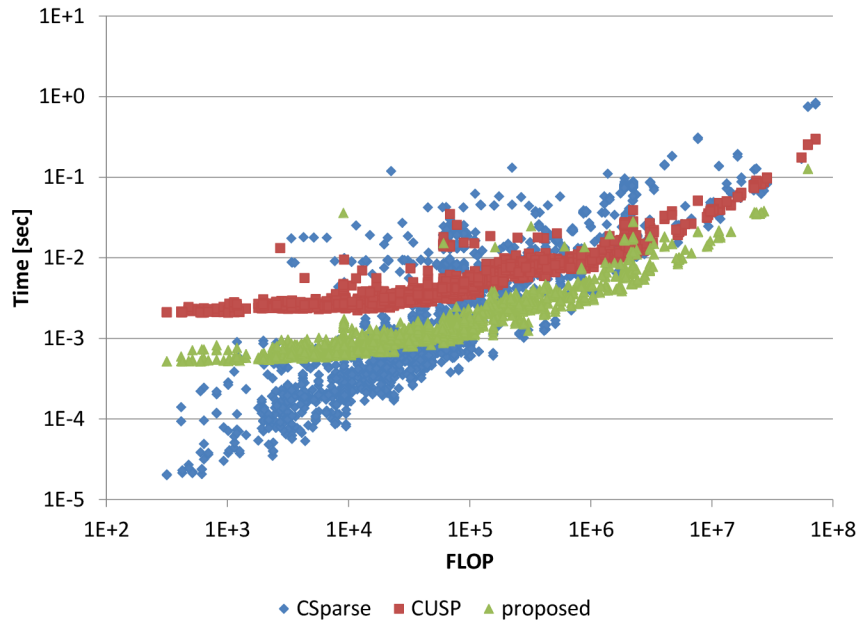


Figure 12.4: Performance scaling comparison on Tesla K40. Note that both axes are logarithmic.

Table 12.1: Matrices from the SNAP subset and CSparse matrix multiplication performance.

Matrix	nnz/row	FLOPs	CSparse time
roadNet-CA	2.807	$22.138 \cdot 10^6$	0.774
web-Google	5.571	$91.665 \cdot 10^6$	5.312
email-Enron	10.020	$72.510 \cdot 10^6$	1.150
amazon0312	7.987	$42.368 \cdot 10^6$	1.659
ca-CondMat	8.081	$5.899 \cdot 10^6$	0.140
p2p-Gnutella31	2.363	$539.035 \cdot 10^3$	0.032
wiki-Vote	12.497	$7.254 \cdot 10^6$	0.082
cit-Patents	4.376	$95.457 \cdot 10^6$	13.414
as-Skitter	13.081	$53.771 \cdot 10^9$	<i>out of RAM</i>

kernels. The memory for the expansion and the product was allocated as outlined in Section 12.3, without any prior knowledge of the size of either. All the calculations were carried out in double precision.

Timing results for the all-to-all product benchmarks are on Figure 12.4. Note that for very small matrices of less than ten thousand FLOPs, CSparse is the fastest. For larger matrices, the proposed implementation takes over. Note that time of CSparse scales linearly with the number of FLOPs, as can be expected from a serial implementation of [75]. The times of the parallelized implementations grow slowly

Table 12.2: GPU sparse matrix multiplication performance comparison on the SNAP subset, the best times are in bold (all times in seconds). The last two columns indicate relative speedup over CSpase and CUSP.

Matrix	GF GTX 680		Tesla K40				
	CUSP	ours	CUSP	ours	MFLOPS	×CSp.	×CUSP
roadNet-CA	0.156	0.199	0.103	0.099	223.662	7.823	1.038
web-Google	0.447	0.433	0.315	0.249	368.356	21.347	1.265
email-Enron	0.360	0.271	0.247	0.173	418.961	6.645	1.429
amazon0312	0.209	0.241	0.141	0.123	344.389	13.488	1.148
ca-CondMat	0.035	0.027	0.024	0.015	394.591	9.347	1.575
p2p-Gnutella31	0.015	0.007	0.008	0.003	190.560	11.304	3.003
wiki-Vote	0.036	0.024	0.025	0.015	482.961	5.482	1.633
cit-Patents	<i>out of RAM</i>		0.497	0.446	214.127	30.089	1.114
as-Skitter	<i>out of RAM⁵</i>						

before the GPU gets saturated, then also scale approximately linearly. Least squares was employed to estimate the saturated costs to 27.7 ms/MFLOP for CSpase, 4.2 ms/MFLOP for CUSP and finally 3.0 ms/MFLOP for the proposed.

A more conventional comparison is presented in Table 12.2. This comparison was performed on the SNAP subset of the University of Florida Sparse Matrix Collection, see Table 12.1 for details about the matrices involved. It contains 9 different classes of matrices, a single matrix was chosen from each of them, much like in the evaluation in [120]. Each of the matrices was multiplied by itself (or in case of rectangular matrices, by its transpose). The proposed solution maintains the best times for most of the matrices, except for *roadNet-CA*, where the number of scalar products per element of the r.h.s. matrix is very low, yielding high thread divergence in the proposed implementation. On smaller matrices such as *p2p-Gnutella31*, CUSP does not scale well and is slower despite the divergence. Reducing this divergence is the subject of the future work.

Note that on *cit-Patents*, both the proposed and CUSP ran out of memory on GTX 680, and on *as-Skitter* there was not enough system memory to perform the multiplication even on the CPU. This is not a principal problem of the algorithm, rather it is an implementation issue. One would only need to add an extra parameter of how many columns of the r.h.s. matrix should be processed at a time

⁵ Note that with only 16 GB of RAM, this matrix is too large even for CSpase: the product would take 26.4 GB.

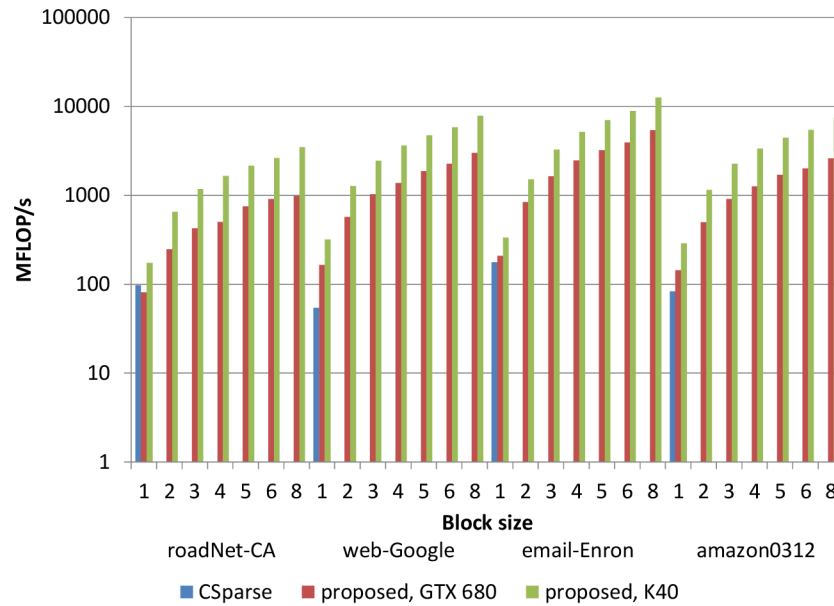


Figure 12.5: Performance scaling comparison of sparse block matrix multiplication on the first four matrices from the SNAP dataset.

(corresponding to the same number of columns of the result), and the CPU would schedule the multiplication as several calls of the original algorithm.

For a synthetic benchmark of the sparse *block* matrix multiplication, the matrices from SNAP were used again. Each element was replaced by a dense block, while the block size was varied between the different tests. The algorithm was slightly modified, to only perform product of the block structure of the matrix, and the actual arithmetics on the dense blocks is performed in the last stage of the algorithm. This significantly reduces the size of the expansion stage, permitting multiplication of even large matrices. The results of this benchmark are on [Figure 12.5](#).

As expected, the proposed implementation exhibits performance increase with increasing block sizes. However, it was discovered that the loop unrolling for known block sizes which was so beneficial on a CPU is not helpful at all on GPU. This is likely because the operation is memory bound and reducing the number of instructions does not yield additional performance. It would likely be more beneficial to use the block size information to choose a tuned implementation when dealing with matrices with multiple block sizes. This remains as a future work.

12.5 CHAPTER SUMMARY

A novel algorithm for sparse matrix multiplication was presented and its extension to sparse block matrices was demonstrated. The algorithm yields on average 329.7 MFLOPS, outperforms CUSP by a factor of 1.53 \times , and outperforms CSpase running on a single CPU by a factor of 13.19 \times . The sparse block matrix multiplica-

tion exhibits further performance scaling with increasing block size, yielding up to 1.26 GFLOPS on Tesla K40 (*email-Enron*, 8×8 blocks). To improve the performance even more, multi-GPU or hybrid CPU-GPU extensions could be implemented. The implementation needs to be improved to handle large matrices by splitting the computation to bands, when the expansion does not fit in the GPU memory at once, as e.g., in the case of *as-Skitter*.

Currently, only constant block size compressed column format (CBC) is supported. This can be extended to variable block size compressed column (VBC), once addressing the possible thread divergence problems. Also, to better integrate with the existing CPU pipelines which use the SSE instruction set, allocation of the product matrix with the proper memory alignment of the blocks needs to be solved. While it is straightforward to align the allocated blocks to pages for the proposed block format, it is not straightforward to also align the blocks inside those pages, in parallel and in a single pass.

ACCELERATING THE NONLINEAR LEAST SQUARES SOLVERS ON GPU

This chapter contains a brief evaluation of the proposed **GPGPU** methods in the context of **NLS** solving. The focus is on timing evaluation, as the precision of the solutions calculated using the proposed methods is not expected to decrease, save for possible rounding errors due to different order of floating-point operations in the parallel implementation and due to a different implementation of the IEEE 754 standard than in the **CPU**.

The evaluation was done on standard **BA** datasets, in the batch mode. This is because in there, the amount of time spent in matrix multiplication in forming the Schur complement is significant compared to the rest of the operations. *Fast & Furious 6* is a bundle adjustment dataset comprising of 160 high-resolution DSLR stills of an open landscape and a highway bridge in Gran Canaria¹. *Guildford Cathedral* comprises of 92 DSLR stills of the Guildford Cathedral² (Surrey, London). *Venice* is a standard bundle adjustment dataset [106] created from an internet collection of 871 photos of a courtyard adjacent to the San Marco square in Venice, Italy.

Some of the tests, including all the **CPU**-only tests, were performed on a machine with a pair of Intel Xeon E5-2470 **CPU**s running at 2.30 GHz and sharing 96 GB of RAM, equipped with a single NVIDIA Tesla K20m **GPU**. Additionally, some tests were performed on an Intel Core i5 **CPU** 661 with 8 GB of RAM and running at 3.33 GHz, equipped with NVIDIA Tesla K40c **GPU**. During the tests, the computers were not running any time-consuming processes in the background. Each test was run several times and the average time was calculated to avoid measurement errors. The turbo boost function of the Xeon **CPU** was disabled for the benchmarks, so as to not make the results dependent on the variations in the temperature.

The results can be seen in [Figure 13.1](#) and [Table 13.1](#). The solutions using a direct solver without the Schur complement are denoted direct- Λ -CS (CSparse [41]), direct- Λ -CM (Cholmod [42]) and direct- Λ -BC (the block Cholesky proposed in [Chapter 5](#)). The times are relatively similar, with CSparse being better in the *Fast & Furious 6* dataset, and the block Cholesky being better in the larger *Venice* dataset. The times with Schur complement, denoted Schur-BC, are improved by about a factor of two. Note that the reported times include also calculation of the Jacobians

¹ Kindly provided by Double Negative Visual Effects, <http://www.dneg.com/>.

² Freely available at <http://cvssp.org/impart/>, upon request.

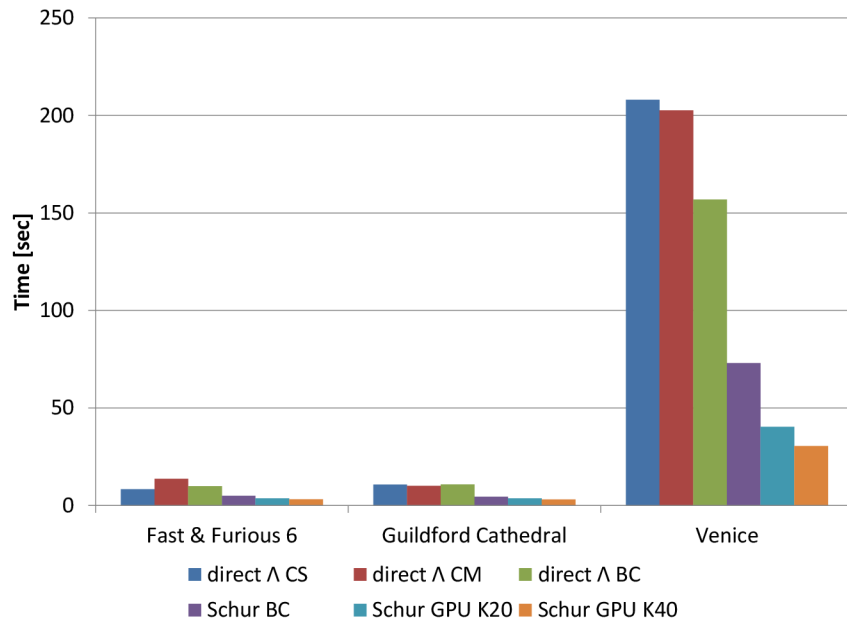


Figure 13.1: GPU-accelerated NLS solving performance on the standard BA datasets.

and other tasks that are the same for all the versions of the algorithm. This makes the perceived speedup slightly smaller than that of the linear solver only (those are reported in Table 8.1).

The GPU-accelerated Schur complement achieves about 50% speedup compared to Schur complement using block Cholesky on the first two smaller datasets, but almost 150% speedup on the larger *Venice* dataset, using Tesla K40. The time required to calculate the Jacobians and update the system is about 70% of the total time for *Fast & Furious 6*, 60% for the *Guildford Cathedral* and 42% for *Venice*, which explains the speedups. To further accelerate the solving, it would be necessary to calculate the Jacobians also on the GPU. Although that would be suitable for a specialized system, it would be difficult to implement generally in an extensible library such as SLAM ++.

The χ^2 errors of all the implementations are basically the same, with differences appearing at the eighth or ninth decimal place. The GPUs have consistently higher error, but the difference is entirely insignificant. Note that all these computations are performed in *double* precision. In context of GPU computing, single precision is more common. However, the NLS solving can be numerically demanding and could easily diverge or produce special numbers if using single precision only. For that reason, the Tesla-class GPUs were used in this comparison. Those are specifically tailored for scientific computation and have more double precision units than gaming or other professional GPUs.

The GPU proves to be a useful tool in the context of small-scale acceleration, such as in the robotics scenarios where the processing needs to be performed in an online fashion. However, the limit of acceleration seems to be low, perhaps with the

Table 13.1: GPU-accelerated NLS solving performance on the standard BA datasets, the best times in bold.

Dataset	Fast & Furious 6	Guildford Cathedral	Venice
direct- Λ -CS	8.337	10.867	208.052
direct- Λ -CM	13.720	10.045	202.669
direct- Λ -BC	9.889	10.742	156.924
Schur-BC	4.916	4.426	73.039
Schur-GPU K20m	3.632	3.646	40.363
Schur-GPU K40c	3.099	3.052	30.503
χ^2 direct- Λ -CS	973514.12	3372728.35	233948930.76
χ^2 direct- Λ -CM	973514.12	3372728.37	233948936.39
χ^2 direct- Λ -BC	973514.12	3372728.35	233948943.53
χ^2 Schur-BC	973514.12	3372728.33	233948938.86
χ^2 GPU K20m	973514.12	3372728.36	233948948.27
χ^2 GPU K40c	973514.14	3372728.35	233948938.13

exception of image processing and other embarrassingly parallel tasks. For large-scale parallelization, distributed processing on CPUs seems to be a better choice, although it presents its own set of challenges.

Part IV

CONCLUSIONS

CONCLUSIONS

The main focus of this thesis was on efficient sparse numerical linear algebra routines with applications in Nonlinear Least Squares (NLS) solving. We selected a particular class of NLS problems that are sparse and exhibit a natural block structure. This block structure was exploited in the implementation of SLAM++, a high performance NLS solver. Having fast arithmetics on block matrices naturally led to the development of more efficient algorithms for incremental matrix factorization and direct solving which would have been impractical or elaborate when using elementwise sparse matrices. GPU acceleration of the key routines on those matrices was also performed. All of the algorithms were rigorously evaluated on standard datasets and compared with similar state of the art implementations.

To summarize, the main contributions of the work presented in this thesis are:

NEW SPARSE BLOCK MATRIX FORMAT: While the format proposed in [Chapter 5](#) is similar to VBR, there are some important improvements. Attention is paid to incremental processing and thus adding new elements to the matrix. When sparse matrices are stored, e.g., in CSC format, this is difficult and typically needs data shuffling – in practice, the matrices are kept in the C00 format for simple modification and then compressed to CSC for numerical operations. Our new data structure takes this overhead away by allowing modification of the matrix structure.

EFFICIENT ARITHMETICS FOR SPARSE BLOCK MATRICES: Although there are some block matrix implementations available today, they are plagued by the extra complexity of the format; intuitively, more nested loops are needed compared with elementwise sparse matrices. In addition, the algorithms for sparse block matrices need to contain extra logic for correctly combining the block layout of the input operands and producing block layout of the output. Such logic is not needed in elementwise sparse matrices. This thesis proposes a highly efficient implementation of arithmetic operations on sparse block matrices, using a novel loop unrolling methodology based on C++ metaprogramming. The results are summarized in [Section 5.3](#).

SPARSE BLOCK MATRIX FACTORIZATIONS: Both batch and incremental variants of the Cholesky factorization were proposed, implemented and benchmarked in this thesis. There are only a few implementations of block matrix factorization to date, which do not offer significant performance advantages and

are not used in practice. The proposed factorization is evaluated on synthetic data in [Section 5.3](#) and also on real robotics problems in [Section 7.8](#) and on computer vision problems in [Section 8.4](#).

EFFICIENT VARIABLE REORDERING STRATEGY FOR INCREMENTAL CHOLESKY FACTORIZATION: The incremental [NLS](#) solver employing the resumed Cholesky algorithm relies on variable reordering to reduce the fill-in. Other implementations perform periodic variable reordering with the fill-in rising uncontrollably in between. A novel incremental variable reordering strategy was proposed in [Section 7.6](#). In addition to this reordering strategy, the new resumed Cholesky factorization algorithm can reuse a part of the factorization that was not affected by the update. This allowed for incremental solvers that superseded the current state of the art.

ANALYSIS OF THE COMPUTATIONAL COMPLEXITIES IN SCHUR COMPLEMENT: For analyzing the numbers of [FLOPs](#), a small library for exactly counting the floating point operations in sparse matrix operations was implemented. It was discovered that using the Schur complement for linear solving, in fact, increases the number of [FLOPs](#) by about a factor of two, rather than maintaining it. This shows the importance of structured memory access patterns. For more details, please refer to [Chapter 8](#).

CLIQUE-BASED ORDERING FOR SCHUR COMPLEMENT: Orderings based on finding the (weighted) Maximum Independent Set ([MIS](#)) and Maximum Independent Clique Set ([MICS](#)) were proposed in [Section 8.1](#) and evaluated with both simple and nested Schur complements on standard datasets. The [MICS](#) ordering can be nested several times, finding relatively large diagonal sections. It was shown how to convert the problem of finding the maximum independent cliques set to finding cliques and subsequently finding an ordinary maximum independent set in a modified graph. The new orderings open possibilities for parallel and hybrid [GPU-CPU](#) processing.

INCREMENTAL SCHUR COMPLEMENT: Similarly to incremental solving in [SLAM](#) problems, an efficient method for incremental solving of Schur-complemented systems was proposed in [Section 8.2](#). It reduces the amount of computation in forming the Schur complement and yields up to an order of magnitude speedup if [GPU](#) acceleration is also available.

SPARSE BLOCK MATRIX FORMULATION OF THE RECURSIVE FORMULA: The recursive formula used for sparse covariance recovery was originally formulated for general sparse matrices. In [Section 9.2](#), we demonstrated that a

blockwise formulation is practical for recovering covariances of multidimensional variables and brings significant performance advantages.

INCREMENTAL COVARIANCE MATRIX UPDATE AND DOWNDATE: A novel approach for incrementally updating a previously calculated covariance of the variables in an **NLS** solver is proposed in [Section 9.2](#). It proved to be about two orders of magnitude faster than the recursive formula implementations in the existing solvers. Furthermore, an elegant method of update by downdate is proposed, which allows maintaining the covariance matrix independently from the nonlinear solver.

SPARSE COVARIANCE RECOVERY FOR SCHUR COMPLEMENTED SYSTEMS:

Following the work on covariance recovery in general **NLS** systems, formulas for recovering covariances in Schur complemented systems were derived in [Section 9.3](#) and their performance tested. It follows that the covariance of the variables inside the Schur complement can be recovered simply by inverting the Schur complement, e.g. using the recursive formula. In [BA](#), this is suitable for finding the covariances in the reduced camera system but it was found that calculating the Schur complement of the landmarks generates large dense matrix, taking hundreds of GB or even TB and is impractical. An alternative formulation was proposed using *sparse sparse* backsubstitution, significantly outperforming the recursive formula on the original system.

The algorithms for covariance recovery were field-tested in a European project, IMPART, by the Double Negative Visual Effects¹ Company. They integrated our implementation of the proposed algorithms in their in-house tool Jigsaw and applied them for estimation of the quality of 3D reconstructions captured on a film set for the support of special effects. This significantly improved the existing workflow, enabled timely quality control and aided the film crew in creative decision making.

FAST GPU SORTING KERNEL: Based on the requirements established by evaluating the proposed **NLS** solver implementation on a **CPU**, fast sorting was identified as one of the key algorithms required for implementing accelerated matrix operations on a **GPU**. Sorting is employed in sparse matrix compression, multiplication and transposition and can also facilitate load balancing in other general tasks. Rather than using the platform-dependent **CUDA** library, the implementation proposed in [Chapter 11](#) uses OpenCL, which lacks sorting implementations on a par with those in **CUDA**. The radix sort algorithm was implemented, with improvements in parallel histogram calculation.

¹ <http://www.dneg.com>

FAST GPU SPARSE MATRIX MULTIPLICATION KERNEL: Used especially by the Schur complement in both batch and incremental variants but also in the incremental covariance update, sparse matrix - matrix multiplication is one remaining bottleneck that cannot be accelerated using the readily available implementations because they are slower than the proposed sparse block schemes on a CPU. Sparse block matrix implementation on a GPU is described and evaluated in Chapter 12, showing substantial performance improvements over similar GPU implementations.

14.1 FUTURE WORK

The sparse block matrix factorizations presented here, despite being highly competitive and outperforming even state of the art implementations such as Cholmod [42], are just the first attempts with hardly any performance tuning. It is possible to employ dense block vectors to accumulate dot products between block columns with different sparsity patterns (as described e.g., in [75]), rather than using the ordered merge algorithm. The memory alignment is currently performed on all of the blocks, likely hurting performance when small blocks are present. It is straightforward to add a memory alignment policy that would disable alignment of those small blocks, based either on expert knowledge or auto-tuning. A number of other low-level improvements and optimizations could be implemented, including also compile-time optimizations.

Furthermore, the proposed block matrix factorizations are simplicial. Their supernodal forms can be implemented to gain significantly better performance. Efficient multifrontal or parallel CPU implementations would also yield a considerable speedup.

To extend the applicability of the proposed methods, other decompositions than Cholesky should be implemented. While being computationally efficient, it is only applicable to symmetric, positive definite matrices. Block-based pivoting was discussed in Section 5.2.5. An efficient pivoting strategy is needed for implementation of LU and QR factorizations, which can be used on general square or rectangular matrices, where it directly affects the numerical stability of the factorization and also affects the resulting fill-in.

The implementation of the specialized block matrix kernels expects a complete, exhaustive list of block sizes that can occur in the input – it is fully specialized. It would be very simple to specialize it only partially – to handle matrices with blocks of sizes that are not on the list, i.e., specifying only a few of the most common block sizes to be processed by the specialized dense kernels while the few blocks of different sizes would be handled using a generic variable-size dense kernel. This would reduce the depth of the block-size decision tree on matrices

that contain many different block sizes and at some point would outperform the fully specialized version.

In the incremental Cholesky factorization, a constrained fill-reducing ordering on a section of the matrix is employed. The whole section is then refactorized using the resumed Cholesky algorithm. It is possible to track the variable dependencies in the factorization and only recalculate those columns that are affected by the update. Alternatively, the Bayes Tree algorithm demonstrates that it is possible to reorder variables in an already factorized matrix. It should be possible to reorder the variables so as to best accommodate the update (e.g., by ordering the affected variables last) and to reduce the fill-in at the same time.

The [MIS](#) and [AMICS](#) orderings for the Schur complement only focus on maximizing the size of the diagonal section. While that leads to a reduction in the size of the Schur complement and thus memory savings, the variables inside each diagonal block and the diagonal blocks themselves can be arbitrarily reordered. This can be used to improve memory access patterns, possibly also saving some fill-in in the Schur complement.

The block matrix kernels on the [GPU](#) are designed with small blocks in mind, which means that the individual blocks have to fit into the shared memory. It would be simple to also design an implementation for very large blocks that do not fit, and slightly more challenging to design an implementation that allows mixtures of both small and large blocks while being able to facilitate reasonable load balancing. Applications of block matrices with very large blocks can be found e.g. in computational chemistry.

The algorithms described in this thesis were implemented with a single-process model in mind and could also be extended to [GPU-CPU](#) hybrid or distributed computing and out-of-core processing. The derivatives are now calculated on the [CPU](#) and consume a significant portion of the time budget. If the analytic expressions for the derivatives are known, it is straightforward to offload this computation onto the [GPU](#). Expression templates and concurrent evaluation of the expression dependency trees could also increase performance.

Part V

APPENDIX

APPENDIX

The purpose of this appendix is to briefly revise the C++ code constructs used in unrolling the loops in the fixed block size implementation described in [Chapter 5](#). It uses fairly advanced constructs and the readers who are not familiar with them are kindly referred to [5].

A.1 TYPELIST

Typelist is a construct which can be used to store structured information which is constant at compile time. A basic typelist declaration can look like this:

This is C++98; the newer versions of the standard bring little advantage, despite the variadic templates introduced in C++11.

Listing A.1: A basic typelist.

```

1: template <class CHead, class CTail>
2: class CTypelist {
3: public:
4:     typedef CHead _TyHead; // head type
5:     typedef CTail _TyTail; // tail type (another CTypelist or CTypelistEnd)
6: };
7:
8: class CTypelistEnd {}; // end marker type, also an empty typelist by itself
9:
10: #define MkTypelist(...) BuildTypelistSomehow<__VA_ARGS__>::Result

```

This enables us to declare recursive structures which form lists, where the elements are types. The `MkTypelist` macro is for convenience only and there are several ways to implement it, which are not disclosed here for the sake of space. An interested reader can refer to [5]. To store (compile-time constant) data in a type, templates can be used again:

Listing A.2: Storing values in a typelist.

```

1: template <int value>
2: struct CCTScalar { // a compile-time scalar class
3:     enum { n_value = value }; // to be able to read the stored value
4: };
5:
6: typedef MkTypelist(CCTScalar<3>, CCTScalar<5>, CCTScalar<7>) BA_var_dims;

```

The last line declares a typelist with three elements which correspond to dimensions of variables in a hypothetical Bundle Adjustment (BA) problem (3D landmarks, 5D camera intrinsic parameters and 7D camera poses). These would be used to specialize algorithms for those given sizes.

A.2 COMPILER-GENERATED DECISION TREE

The idea behind the fixed block size implementation is to make the compiler build a decision tree for the block sizes and perform arithmetics inside each leaf, with the loops unrolled. To build a decision tree, a few operations on the typelist are needed: calculating its length (easily accomplished using a recursive template which increments a counter for each recursion until it finds `CTypeListEnd`), getting an element by index (a very similar template, this time decrementing an index and returning the current head type once it reaches zero) and sorting a typelist.

Sorting is the only non-trivial operation, which was implemented using a bubble sort. While not the most efficient algorithm for general arrays, it is quite suitable for linked lists – and thus also typelists. In any case, the sorting takes place at compile time.

To build the decision tree, a common skeleton class is used, which calculates the positions and values (types) of pivots, as illustrated in [Listing A.3](#). In order to save space, it is assumed here that the elements of the tree are specializations of `CCTScalar` and it is thus known how to compare them to the needle (*needle* refers to the item being searched for). However, it is not difficult to add the needle type and the needle to pivot comparison algorithm to the list of decision tree template parameters, to make it fully general.

This uses C++14 in order to save a few extra lines of code; it is very easy to implement enable_if in C++98 as well.

To use such a decision tree skeleton is surprisingly easy, as illustrated in [Listing A.4](#). In here, the function `DTEExample` uses a decision tree to invoke the function operator of functor `CDTEExampleFunctor` which in turn gets the run-time value 5 as a compile-time constant. Although this may seem like a much ado to do just that, it is the enabling component which makes sparse block matrices practical. At the same time, `CDTEExampleFunctor` could carry data in its member variables, such as references to the input and output vectors or matrices to do arithmetics with.

Note that this example relies on the typelist to be sorted (otherwise the decision tree would not work) and in the implementation, one more layer which takes care of the sorting and making sure there are no duplicate entries is employed. The result is a nicely verbose construct `CWrap::In_DecisionTree<ListOfSizes>(int size, Functor f)`. For the operations on block matrices, the decision trees can be over column width, row heights or only some row heights which can occur in a column of a given width.

Listing A.3: Decision tree skeleton.

```

1: template <class CList>
2: class CMakeDecisionTreeSkeleton {
3: public:
4:     template <int n_begin, int n_length>
5:     class CSkeleton {
6:     public:
7:         enum {
8:             n_half = (n_length + 1) / 2, // half of the length (round up)
9:             n_pivot = n_begin + n_half - 1, // index of pivot
10:            b_leaf = false // leaf flag
11:        };
12:
13:        typedef typename CTypelistGet<CList, n_pivot>::_TyResult _TyPivot;
14:
15:        typedef CSkeleton<n_begin + n_half, n_length - n_half> _TyLeft;
16:        typedef CSkeleton<n_begin, n_half> _TyRight;
17:        // left and right subtrees
18:
19:        static bool b_Left_of_Pivot(int needle)
20:        {
21:            return _TyPivot::n_value < needle; // or use custom comparator
22:        }
23:    };
24:
25:    template <int n_begin>
26:    class CSkeleton<n_begin, 1> {
27:    public:
28:        enum {
29:            n_pivot = n_begin, // index of pivot
30:            b_leaf = true // leaf flag
31:        };
32:
33:        typedef typename CTypelistGet<CList, n_pivot>::_TyResult _TyPivot;
34:
35:        static inline bool b_Equals_Pivot(int needle)
36:        {
37:            return _TyPivot::n_value == needle; // or use custom comparator
38:        }
39:    };
40:
41:    typedef CSkeleton<0, CTypelistLength<CList>::n_result> _TyRoot;
42: };

```

Listing A.4: Wrapping an algorithm in a decision tree.

```

1: template <class CList>
2: class CDecisionTree {
3:     typedef typename CMakeDecisionTreeSkeleton<CList>::_TyRoot _TyRoot;
4:
5:     template <class CNode, class CContext, typename =
6:         std::enable_if_t<CNode::b_leaf> > // version for leaves
7:     static void Recurse(int needle, CContext context)
8:     {
9:         assert(CNode::b_Equals_Pivot(needle)); // the needle is present
10:        context.template operator ()<CNode::_TyPivot::_n_value>();
11:    }
12:
13:    template <class CNode, class CContext, typename =
14:        std::enable_if_t<!CNode::b_leaf> > // version for internal nodes
15:    static void Recurse(int needle, CContext context)
16:    {
17:        if(CNode::b_Left_of_Pivot(needle))
18:            Recurse<typename CNode::_TyLeft>(needle, context);
19:        else
20:            Recurse<typename CNode::_TyRight>(needle, context);
21:    }
22:
23: public:
24:     template <class CContext>
25:     static void Run(int n_size, CContext context)
26:     {
27:         Recurse<_TyRoot>(n_size, context);
28:     }
29: };
30:
31: struct CDTEExampleFunctor {
32:     template <int block_size>
33:     void operator ()() const
34:     {
35:         assert(block_size == 5); // now it is a compile time constant
36:     }
37: };
38:
39: void DTEExample()
40: {
41:     CDecisionTree<BA_var_dims>::Run(5, CDTEExampleFunctor());
42: } // the 5 is passed as a run-time variable

```

Listing A.5: Fixed block size matrix vector multiplication.

```

1: template <class CBlockMatrixTypelist>
2: struct GAXPY {
3:     template <int n_col_width> // function object can be a template too
4:     struct CInnerLoop {
5:         const double *m_A, *m_x; // inputs
6:         double *m_y; // input / output
7:
8:         CInnerLoop(const double *A, double *y, const double *x); // ...
9:
10:        template <int n_row_height>
11:        void operator ()()
12:        {
13:            for(int c = 0; c < n_col_width; ++ c) { // unrolls
14:                for(int r = 0; r < n_row_height; ++ r) // unrolls
15:                    m_y[r] += m_A[r + c * n_col_width] * m_x[c];
16:            } // y = A * x + y
17:        }
18:    };
19:
20:    struct COuterLoop {
21:        const TColumn &m_block_col;
22:        double *m_y;
23:        const double *m_x;
24:        const std::vector<TRow> &m_block_row_list;
25:
26:        COuterLoop(const TColumn &block_col, double *y, const double *x,
27:            const std::vector<TRow> &block_row_list); // ...
28:
29:        template <int n_col_width>
30:        void operator ()()
31:        {
32:            const double *p_x = &m_x[m_block_col.n_first_column];
33:            FOR_EACH(block in m_block_col) {
34:                const TRow &block_row = m_block_row_list[block.row];
35:                double *p_y = &m_y[block_row.n_first_row];
36:
37:                CWrap::In_RowHeight_DecisionTree_Given_ColumnWidth<
38:                    CBlockMatrixTypelist, n_col_width>(block_row.n_height,
39:                    CInnerLoop<n_col_width>(block.data, p_y, p_x));
40:            } // wrap the inner loop in the row height decision tree
41:        } // for row heights which are possible in this column
42:    };

```

Listing A.6: Fixed block size matrix vector multiplication (continued).

```
1:     static void Run(const CSparseBlockMatrix &matrix,  
2:         double *y, const double *x)  
3:     {  
4:         FOR_EACH(block_col in matrix.BlockCol_List()) {  
5:             CWrap::In_ColumnWidth_DecisionTree<CBlockMatrixTypelist>(  
6:                 block_col.n_width, COuterLoop(block_col, y, x,  
7:                 matrixl.BlockRow_List()));  
8:             // wrap the outer loop in a column width decision tree  
9:         }  
10:    }  
11: };
```

The final example in [Listing A.5](#) is the pseudocode for the sparse block matrix - dense vector multiplication routine. The code is split into three sections: `GAXPY::CInnerLoop` containing the body of the inner loop which calculates product of a single dense block with the vector, `GAXPY::COuterLoop` with the body of the outer loop which iterates over nonzero blocks in each block column and finally the outer loop itself in `GAXPY::Run` which iterates over the block columns in the matrix. Note that the constructors of the loop classes were omitted to save space; they only copy their arguments to the corresponding member variables.

Note that all the loops inside the function operator of `CInnerLoop` can be unrolled, which is the key to the high performance arithmetics routines for sparse block matrices. It is even possible to write specializations for specific dimensions, which use handcrafted assembly code, e.g. to use [SIMD](#) instructions. The proposed sparse block matrix implementation uses this technique throughout and implements all the matrix routines in this manner.

BIBLIOGRAPHY

- [1] Prabhakar Agarwal, Gian Diego Tipaldi, Luciano Spinello, Cyrill Stachniss, and Wolfram Burgard. Robust map optimization using dynamic covariance scaling. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 62–69. IEEE, 2013.
- [2] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, M Joshi, and P Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM J. of Research and Development*, 39(5):575–582, 1995.
- [3] S. Agarwal and K. Mierle. Ceres solver. <http://ceres-solver.org/>, 2012.
- [4] Sameer Agarwal, Noah Snavely, Ian Simon, Steven M. Seitz, and Richard Szeliski. Building rome in a day. In *Intl. Conf. on Computer Vision (ICCV)*, Kyoto, Japan, 2009.
- [5] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001. ISBN 978-0-201-70431-0.
- [6] P. Amestoy, T. A. Davis, and I. S. Duff. Amd, an approximate minimum degree ordering algorithm). *ACM Trans. Math. Software*, 30(3):381–388, September 2004.
- [7] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. An approximate minimum degree ordering algorithm. *SIAM J. on Matrix Analysis and Applications*, 17(4):886–905, 1996.
- [8] 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*, volume 9. SIAM, 1987. ISBN 978-0-898-71447-0.
- [9] Anita Araneda and Alvaro Soto. Statistical inference in mapping and localization for mobile robots. In *Advances in Artificial Intelligence–IBERAMIA 2004*, pages 545–554. Springer Heidelberg, 2004.
- [10] Tim Bailey, Juan Nieto, and Eduardo Nebot. Consistency of the FastSLAM algorithm. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 424–429. IEEE, 2006.
- [11] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software

- libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.
- [12] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015. URL <http://www.mcs.anl.gov/petsc>.
- [13] Kenneth E Batchner. Sorting networks and their applications. In *Proc. of the AFIPS Conf. Spring Joint Computer Conference (SJCC)*, pages 307–314. ACM, 1968.
- [14] C. Beall, B.J. Lawrence, V. Ila, and F. Dellaert. 3D reconstruction of underwater structures. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2010.
- [15] Albert E Beaton and John W Tukey. The fitting of power series, meaning polynomials, illustrated on band-spectroscopic data. *ASQ/ASA Technometrics*, 16(2):147–185, 1974.
- [16] Nathan Bell, Steven Dalton, and Luke N Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM J. on Sci. Computing*, 34(4): C123–C152, 2012.
- [17] Christopher M Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995. ISBN 978-0-198-53864-6.
- [18] A. Björck. *Numerical methods for least squares problems*. SIAM, 1996. ISBN 978-0-898-71360-2.
- [19] Guy E Blelloch. *Vector models for data-parallel computing*, volume 75. MIT Press, 1990. ISBN 978-0-262-02313-9.
- [20] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graphics*, 22(3):917–924, 2003.
- [21] M.C. Bosse, P.M. Newman, J.J. Leonard, and S. Teller. Simultaneous localization and map building in large-scale cyclic environments using the Atlas framework. *Intl. J. of Robotics Research*, 23(12):1113–1139, Dec 2004.
- [22] DR Bowler, T Miyazaki, and MJ Gillan. Parallel sparse matrix multiplication for linear scaling electronic structure calculations. *Computer Physics Commun.*, 137(2):255–273, 2001.

- [23] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 233–244. ACM, 2009.
- [24] Aydin Buluç and John R Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*, pages 503–510. IEEE, 2008.
- [25] Richard H Byrd, Robert B Schnabel, and Gerald A Shultz. Approximate solution of the trust region problem by minimization over two-dimensional subspaces. *Mathematical Programming*, 40(1–3):247–263, 1988.
- [26] Martin Byröd and Kalle Åström. Conjugate gradient bundle adjustment. In *Eur. Conf. on Computer Vision (ECCV)*, pages 114–127. Springer Heidelberg, 2010.
- [27] Luca Carlone, Zsolt Kira, Chris Beall, Vadim Indelman, and Frank Dellaert. Eliminating conditionally independent sets in factor graphs: A unifying perspective based on smart factors. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 4290–4297. IEEE, 2014.
- [28] Luca Carlone, Roberto Tron, Kostas Daniilidis, and Frank Dellaert. Initialization techniques for 3D SLAM: a survey on rotation estimation and its use in pose graph optimization. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2015.
- [29] S. Carney, M.A. Heroux, G. Li, and K. Wu. A revised proposal for a sparse BLAS toolkit. Technical report, SPARKER Working Note, 1994.
- [30] P. M. Cassereau, D. H. Staelin, and G. de Jager. Encoding of images based on a lapped orthogonal transform. *IEEE Trans. Commun.*, 37(2):189–193, Feb 1989. ISSN 0090-6778. doi: 10.1109/26.20089.
- [31] Philippe Michel Cassereau. *A new class of optimal unitary transforms for image processing*. PhD thesis, MIT, 1985.
- [32] Denis Chekhlov, Mark Pupilli, Walterio Mayol-Cuevas, and Andrew Calway. Real-time and robust monocular SLAM using predictive multi-resolution descriptors. In *Intl. Symp. on Advances in Visual Computing*, pages 276–285. Springer Heidelberg, 2006.
- [33] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Software*, 35(3):22, 2008.

- [34] Jaeyoung Choi, David W Walker, and Jack J Dongarra. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.
- [35] Kok Seng Chong and Lindsay Kleeman. Feature-based mapping in real, large scale environments using an ultrasonic array. *Intl. J. of Robotics Research*, 18(1):3–19, 1999.
- [36] Gábor Csárdi and Tamás Nepusz. The igraph software package for complex network research. *International Journal of Complex Systems – Computing, Sensing and Control*, 1695(5):1–9, 2006.
- [37] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In *Sparse Matrices and Their Applications*, pages 157–166. Springer US, 1972.
- [38] Steven Dalton, Nathan Bell, and Luke Olson. Optimizing sparse matrix-matrix multiplication for the GPU. Technical report, Technical Report, 2013.
- [39] T.A. Davis. The university of florida sparse matrix collection. In *NA Digest*. Citeseer, 1994.
- [40] Timothy A. Davis. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software*, 30(2):196–199, June 2004. ISSN 0098-3500. doi: 10.1145/992200.992206.
- [41] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. SIAM, 2006. ISBN 0-89871-613-6.
- [42] Timothy A. Davis and William W. Hager. Modifying a sparse cholesky factorization, 1997.
- [43] A.J. Davison and D.W. Murray. Simultaneous localization and map-building using active vision. *IEEE Trans. Pattern Anal. Machine Intell.*, 24(7):865–880, Jul 2002.
- [44] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM J. on Computing*, 10(4):657–675, 1981.
- [45] F. Dellaert and M. Kaess. Square Root SAM: Simultaneous localization and mapping via square root information smoothing. *Intl. J. of Robotics Research*, 25(12):1181–1203, Dec 2006.
- [46] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson. An extended set of basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Software*, 14(1):18–32, 1988.

- [47] J.J. Dongarra, J. Du Croz, S. Hammarling, and I.S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16(1):1–17, 1990.
- [48] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.
- [49] Iain Du and Michele Marrone. A proposal for user level sparse BLAS. Technical report, Rutherford Appleton Laboratory, Oxfordshire and CERFACTS, Toulouse and IBM Semea, Cagliari, 1992.
- [50] I. S. Duff and J. K. Reid. An implementation of Tarjan’s algorithm for the block triangularization of a matrix. *ACM Trans. Math. Software*, 4(2):137–147, 1978. ISSN 0098–3500. doi: 10.1145/355780.355785.
- [51] Iain S Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Software*, 7(3):315–330, 1981.
- [52] Iain S Duff, Roger G Grimes, and John G Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15(1):1–14, 1989.
- [53] Iain S Duff, John K Reid, and Jennifer A Scott. The use of profile reduction algorithms with a frontal code. *Intl. J. for Numerical Methods in Eng.*, 28(11):2555–2568, 1989.
- [54] HF Durrant-Whyte, MWMG Dissanayake, and PW Gibbens. Toward deployment of large scale simultaneous localisation and map building (SLAM) systems. In *Proc. of the Intl. Symp. of Robotics Research (ISRR)*, volume 9, pages 161–168, 2000.
- [55] C. Engels, H. Stewénus, and D. Nistér. Bundle adjustment rules. In *Symposium on Photogrammetric Computer Vision*, pages 266–271, Sep 2006.
- [56] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation–ISAAC 2010*, pages 403–414. Springer Heidelberg, Berlin, 2010. ISBN 978-3-642-17517-6. doi: 10.1007/978-3-642-17517-6_36.
- [57] R.M. Eustice, H. Singh, J.J. Leonard, and M.R. Walter. Visually mapping the RMS Titanic: Conservative covariance estimates for SLAM information filters. *Intl. J. of Robotics Research*, 25(12):1223–1242, Dec 2006.
- [58] Gordon C Everstine. A comparasion of three resequencing algorithms for the reduction of matrix profile and wavefront. *Intl. J. for Numerical Methods in Eng.*, 14(6):837–853, 1979.

- [59] Jeanette F. Intel math kernel library. reference manual. Technical report, Intel Corporation, Santa Clara, USA, 630813-054US, 2009. URL <http://software.intel.com/intel-mkl/>.
- [60] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pages 133–137. ACM, 2004.
- [61] Xiao-Shan Gao, Xiao-Rong Hou, Jianliang Tang, and Hang-Fei Cheng. Complete solution classification for the perspective-three-point problem. *IEEE Trans. Pattern Anal. Machine Intell.*, 25(8):930–943, 2003.
- [62] Andreas Geiger, Julius Ziegler, and Christoph Stiller. StereoScan: Dense 3D reconstruction in real-time. In *IEEE Intelligent Vehicles Symp. (IV)*, 2011.
- [63] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The KITTI dataset. *Intl. J. of Robotics Research*, 2013.
- [64] Alan George. Nested dissection of a regular finite element mesh. *SIAM J. on Numerical Anal.*, 10(2):345–363, 1973.
- [65] Roman Geus and Stefan Röllin. Towards a fast parallel sparse symmetric matrix–vector multiplication. *Proc. of the Intl. Conf. on Parallel Computing (ParCo)*, 27(7):883–896, 2001.
- [66] Norman E. Gibbs. Algorithm 509: A hybrid profile reduction algorithm [F1]. *ACM Trans. Math. Software*, 2(4):378–387, December 1976. ISSN 0098-3500. doi: 10.1145/355705.355713.
- [67] Mark S Gockenbach. *Understanding and implementing the finite element method*. SIAM, 2006. ISBN 978-0-898-71614-6.
- [68] Gene H Golub and Robert J Plemmons. Large-scale geodetic least-squares adjustment by dissection and orthogonal decomposition. *Linear Algebra Appl.*, 34:3–28, 1980.
- [69] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Software*, 35(1):4, 2008.
- [70] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUP-eraSort: high performance graphics co-processor sorting for large database management. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 325–336. ACM, 2006.

- [71] G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard. A tree parameterization for efficiently computing maximum likelihood maps using gradient descent. In *Robotics: Science and Systems (RSS)*, Jun 2007.
- [72] Giorgio Grisetti, Rainer Kummerle, Cyrill Stachniss, Udo Frese, and Christoph Hertzberg. Hierarchical optimization on manifolds for online 2D and 3D mapping. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 273–278. IEEE, 2010.
- [73] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.
- [74] John Gunnels, Calvin Lin, Greg Morrow, and Robert Van De Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proc. of the First Merged Intl. Parallel Processing Symp. and Symp. on Parallel and Distributed Processing (IPPS/SPDP)*, pages 110–116. IEEE, 1998.
- [75] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Software*, 4(3):250–269, 1978.
- [76] JKL Ha, Jens Krüger, and Cláudio T Silva. Implicit radix sorting on GPUs. In Matt Pharr, Randima Fernando, and Tim Sweeney, editors, *GPU Gems*, volume 2. NVIDIA Corporation, Santa Clara, CA, 2010.
- [77] William W Hager. Updating the inverse of a matrix. *SIAM Review*, 31(2):221–239, 1989.
- [78] Frank R Hampel. Robust estimation: A condensed partial survey. *Probability Theory and Related Fields*, 27(2):87–104, 1973.
- [79] A. Handa, M. Chli, H. Strasdat, and A. J Davison. Scalable active matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2010.
- [80] Sebastian Haner and Anders Heyden. Covariance propagation and next best view planning for 3d reconstruction. In *Eur. Conf. on Computer Vision (ECCV)*, pages 545–556, Italy, October 2012.
- [81] Florian Hecht, Yeon Jin Lee, Jonathan R. Shewchuk, and James F. O’Brien. Updated sparse cholesky factors for corotational elastodynamics. *ACM Trans. Graphics*, 31(5):1–13, October 2012. Presented at SIGGRAPH 2012.
- [82] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. URL <http://thrust.github.io/>. Version 1.7.0.
- [83] Paul W Holland and Roy E Welsch. Robust regression using iteratively reweighted least-squares. *Communications in Statistics – Theory and Methods*, 6(9):813–827, 1977.

- [84] Steven Holmes, Georg Klein, and David W Murray. A square root unscented kalman filter for visual monoSLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 3710–3716. IEEE, 2008.
- [85] A. Howard and N. Roy. The robotics data set repository (Radish), 2003. URL <http://radish.sourceforge.net/>.
- [86] Guoquan P Huang, Anastasios Mourikis, Stergios Roumeliotis, et al. On the complexity and consistency of UKF-based SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 4401–4408. IEEE, 2009.
- [87] Shoudong Huang, Heng Wang, Udo Frese, and Gamini Dissanayake. On the number of local minima to the point feature based SLAM problem. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 2074–2079. IEEE, 2012.
- [88] Peter J Huber. *Robust statistics*. Springer Heidelberg, 2011. ISBN 978-0-470-12990-6.
- [89] V. Ila, J. M. Porta, and J. Andrade-Cetto. Information-based compact Pose SLAM. *IEEE Trans. Robotics*, 26(1):78–93, 2010.
- [90] Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Computational Science–ICCS 2001*, pages 127–136. Springer Heidelberg, 2001.
- [91] Vadim Indelman, Richard Roberts, Chris Beall, and Frank Dellaert. Incremental light bundle adjustment. In *British Machine Vision Conf. (BMVC)*, pages 134.1–134.11. BMVA Press, 2012. ISBN 1-901725-46-4. doi: 10.5244/C.26.134.
- [92] S. Julier, J. Uhlmann, and H.F. Durrant-Whyte. A new method for the non-linear transformation of means and covariances in filters and estimators. *IEEE Trans. Automat. Contr.*, 45(3):477–482, Mar 2000. ISSN 0018-9286. doi: 10.1109/9.847726.
- [93] M. Kaess and F. Dellaert. Covariance recovery from a square root information matrix for data association. *Robotics and Autonomous Syst.*, 2009.
- [94] M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Fast incremental smoothing and mapping with efficient data association. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 1670–1677, Rome, Italy, April 2007. ISBN 1-42440-601-3.
- [95] M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Incremental smoothing and mapping. *IEEE Trans. Robotics*, 24(6):1365–1378, Dec 2008.

- [96] M. Kaess, V. Ila, R. Roberts, and F. Dellaert. The Bayes tree: An algorithmic foundation for probabilistic robot mapping. In *Intl. Workshop on the Algorithmic Foundations of Robotics*, Dec 2010.
- [97] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. Leonard, and F. Dellaert. iSAM2: Incremental smoothing and mapping with fluid relinearization and incremental variable reordering. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.
- [98] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert. iSAM2: Incremental smoothing and mapping using the Bayes tree. *Intl. J. of Robotics Research*, 31:217–236, February 2011.
- [99] Anders Karlsson, Jon Bjärkefur, Joakim Rydell, and Christina Grönwall. Smoothing-based submap merging in large area SLAM. In *Image Analysis–SCIA 2011*, Lecture Notes in Computer Sci., pages 134–145. Springer Heidelberg, 2011.
- [100] Chanki Kim, Rathinasamy Sakthivel, and Wan Kyun Chung. Unscented Fast-SLAM: a robust and efficient solution to the SLAM problem. *IEEE Trans. Robotics*, 24(4):808–820, 2008.
- [101] Peter Kipfer and Rüdiger Westermann. Improved GPU sorting. In Matt Pharr, Randima Fernando, and Tim Sweeney, editors, *GPU Gems*, volume 2, chapter 46, pages 733–746. NVIDIA Corporation, Santa Clara, CA, 2005.
- [102] Peter Kipfer, Mark Segal, and Rüdiger Westermann. UberFlow: a GPU-based particle engine. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pages 115–122. ACM, 2004.
- [103] Donald Ervin Knuth. *The art of computer programming. 1, (1973). Fundamental algorithms*. Addison-Wesley, 1973. ISBN 0-201-03801-3.
- [104] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, and R. Vincent. Efficient sparse pose adjustment for 2d mapping. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, October 2010.
- [105] Kurt Konolige. Sparse sparse bundle adjustment. In *British Machine Vision Conf. (BMVC)*, Aberystwyth, Wales, 08/2010 2010.
- [106] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A general framework for graph optimization. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.

- [107] E Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, pages 55–55. ACM, 2001.
- [108] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Software*, 5(3):308–323, 1979.
- [109] J.J. Leonard and P.M. Newman. Consistent, convergent, and constant-time SLAM. In *Intl. Joint Conf. on AI (IJCAI)*, 2003.
- [110] Jan Lienemann, Dag Billger, Evgenii B Rudnyi, Andreas Greiner, and Jan G Korvink. Mems compact modeling meets model order reduction: Examples of the application of arnoldi methods to microsystem devices. In *Proceedings of the NSTI Nanotechnology Conference and Trade Show (Nanotech)*, volume 4, 2004.
- [111] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro Magazine*, 28(2):39–55, 2008.
- [112] Joseph WH Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11(2):141–153, 1985.
- [113] Manolis Lourakis and Antonis Argyros. 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, 2004.
- [114] Manolis IA Lourakis and Antonis A Argyros. SBA: A software package for generic sparse bundle adjustment. *ACM Trans. Math. Software*, 36(1):2, 2009.
- [115] Manolis IA Lourakis, Antonis Argyros, et al. Is Levenberg-Marquardt the most efficient optimization algorithm for implementing bundle adjustment? In *Intl. Conf. on Computer Vision (ICCV)*, volume 2, pages 1526–1531. IEEE, 2005.
- [116] David G Lowe. Object recognition from local scale-invariant features. In *Intl. Conf. on Computer Vision (ICCV)*, volume 2, pages 1150–1157. IEEE, 1999.
- [117] V. Lui and T. Drummond. Image based optimisation without global consistency for constant time monocular visual SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 5799–5806, May 2015. doi: 10.1109/ICRA.2015.7140011.

- [118] Kaj Madsen, Hans Bruun, and Ole Tingleff. Methods for non-linear least squares problems. Technical report, Scientific Computing, Dept. of Informatics and Mathematical Modeling, Technical University of Denmark, 1999.
- [119] Ruben Martinez-Cantin, Josée Castellanos, et al. Unscented SLAM for large-scale outdoor environments. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 3427–3432. IEEE, 2005.
- [120] Kiran Kumar Matam, Siva Rama Krishna Bharadwaj, and Kishore Kothapalli. Sparse matrix multiplication on hybrid CPU+GPU platforms. In *Proc. of the IEEE Intl. Conf. on High Performance Computing, Data, and Analytics (HiPC)*, 2012.
- [121] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Lett.*, 21(02):245–272, 2011.
- [122] Duane G Merrill and Andrew S Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 545–546. ACM, 2010.
- [123] Duane G Merrill III and Andrew Adviser-Grimshaw. *Allocation-oriented algorithm design with application to GPU computing*. PhD thesis, University of Virginia, 2011.
- [124] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Nat. Conf. on Artificial Intelligence (AAAI)*, Edmonton, Alberta, Canada, 2002.
- [125] Jorge J Moré. The Levenberg-Marquardt algorithm: implementation and theory. In *Proc. of the Biennial Conf. on Numerical Analysis*, pages 105–116. Springer Heidelberg, 1978.
- [126] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proc. of the Intl. Conf. on Computer Vision Theory and Applications (VISAPP)*, pages 331–340. INSTICC Press, 2009.
- [127] Karol Myszkowski, Oleg G Okunev, and Toshiyasu L Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–511, 1995.
- [128] Steven Dalton Nathan Bell and Michael Garland. CUSP: Generic parallel algorithms for sparse matrix and graph computations. Technical report, NVIDIA Corporation, 2009. URL <http://cusplibrary.github.com/>.

- [129] J. Neira and J.D. Tardos. Data association in stochastic mapping using the joint compatibility test. *IEEE Trans. Robot. Automat.*, 17(6):890–897, December 2001.
- [130] K. Ni, D. Steedly, and F. Dellaert. Out-of-core bundle adjustment for large-scale 3D reconstruction. In *Intl. Conf. on Computer Vision (ICCV)*, Rio de Janeiro, October 2007.
- [131] K. Ni, D. Steedly, and F. Dellaert. Tectonic SAM: Exact; out-of-core; submap-based SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Rome; Italy, April 2007.
- [132] Kai Ni and Frank Dellaert. Multi-level submap based slam using nested dissection. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2010.
- [133] J. Nieto, H. Guivant, E. Nebot, and S. Thrun. Real time data association for FastSLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2003.
- [134] David Nistér. An efficient solution to the five-point relative pose problem. *IEEE Trans. Pattern Anal. Machine Intell.*, 26(6):756–770, 2004.
- [135] NVIDIA. CUDA C programming guide version 7.5. Technical report, NVIDIA Corporation, Santa Clara, CA, 2015.
- [136] E. Olson, J. Leonard, and S. Teller. Fast iterative alignment of pose graphs with poor initial estimates. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2006.
- [137] Edwin Olson. *Robust and Efficient Robot Mapping*. PhD thesis, Massachusetts Institute of Technology, 2008.
- [138] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search - a log log n search. *Communications of the ACM*, 21(7):550–553, 1978.
- [139] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, page 30. ACM, 1999.
- [140] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Software*, 16(4):303–324, 1990.
- [141] Michael JD Powell. A hybrid method for nonlinear equations. *Numerical methods for nonlinear algebraic equations*, 7:87–114, 1970.

- [142] Sam Prentice and Nicholas Roy. The belief roadmap: Efficient planning in linear POMDPs by factoring the covariance. In *Proc. of the Intl. Symp. of Robotics Research (ISRR)*, pages 293–305. Springer Heidelberg, 2011.
- [143] Timothy J Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.
- [144] M. Self R. Smith and P. Cheeseman. A stochastic map for uncertain spatial relationships. In *Proc. of the Intl. Symp. of Robotics Research (ISRR)*, 1988.
- [145] D Roller, Michael Montemerlo, S Thrun, and Ben Wegbreit. FastSLAM 2.0: an improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *Intl. Joint Conf. on AI (IJCAI)*, 2003.
- [146] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computation. Technical report, Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990.
- [147] Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. *ACM Journal of Experimental Algorithmics (JEA)*, 19:2–2, 2015.
- [148] Chandra Saravanan, Yihan Shao, Roi Baer, Philip N Ross, and Martin Head-Gordon. Sparse matrix multiplications for linear scaling electronic structure calculations in an atom-centered basis set using multiatom blocks. *Journal of Computational Chemistry*, 24(5):618–622, 2003.
- [149] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *IEEE Intl. Symp. on Parallel & Distributed Processing, (IPDPS)*, pages 1–10. IEEE, 2009.
- [150] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for GPU computing. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, volume 2007, pages 97–106. Eurographics Association, 2007.
- [151] Monika Shah and Vibha Patel. An efficient sparse matrix multiplication for skewed matrix on GPU. In *High Performance Computing and Communication & 9th Intl. Conf. on Embedded Software and Systems (HPCC-ICES)*, pages 1301–1306. IEEE, 2012.
- [152] Andrew H Sherman. Algorithms for sparse gaussian elimination with partial pivoting. *ACM Trans. Math. Software*, 4(4):330–338, 1978.

- [153] Dieter Sibley, Christopher Mei, Ian Reid, and Paul Newman. Adaptive relative bundle adjustment. In *Robotics: Science and Systems (RSS)*, volume 32, page 33, 2009.
- [154] Erik Sintorn and Ulf Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *J. of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.
- [155] Noah Snavely, Steven M Seitz, and Richard Szeliski. Photo tourism: exploring photo collections in 3D. *ACM Trans. Graphics*, 25(3):835–846, 2006.
- [156] Richard A Snay. Reducing the profile of sparse symmetric matrices. *Bulletin Géodésique*, 50(4):341–352, 1976.
- [157] Stefano Soatto and Roger Brockett. Optimal structure from motion: Local ambiguities and global estimates. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 282–288. IEEE, 1998.
- [158] Pavel Solin, Karel Segeth, and Ivo Dolezel. *Higher-order finite element methods*. CRC Press, 2004. ISBN 978-1-584-88438-5.
- [159] Hauke Strasdat. *Local Accuracy and Global Consistency for Efficient Visual SLAM*. PhD thesis, Imperial College London, UK, 2012.
- [160] Hauke Strasdat, José MM Montiel, and Andrew J Davison. Visual SLAM: why filter? *Image and Vision Computing*, 30(2):65–77, 2012.
- [161] James O Street, Raymond J Carroll, and David Ruppert. A note on computing robust regression estimates via iteratively reweighted least squares. *ASA The American Statistician*, 42(2):152–154, 1988.
- [162] Catherine Stuart. Robust regression. Technical report, Department of Mathematical Sciences, Durham University, UK, 2011.
- [163] Niko Sünderhauf and Peter Protzel. Switchable constraints for robust pose graph SLAM. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1879–1884. IEEE, 2012.
- [164] Heidi K Thornquist, Eric R Keiter, Robert J Hoekstra, David M Day, and Erik G Boman. A parallel preconditioning strategy for efficient transistor-level circuit simulation. In *IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 410–417. IEEE, 2009.
- [165] S. Thrun, Y. Liu, D. Koller, A.Y. Ng, Z. Ghahramani, and H. Durrant-Whyte. Simultaneous localization and mapping with sparse extended information filters. *Intl. J. of Robotics Research*, 23(7–8):693–716, 2004.

- [166] Sebastian Thrun and Yufeng Liu. Multi-robot SLAM with sparse extended information filters. In *Proc. of the Intl. Symp. of Robotics Research (ISRR)*, pages 254–266. Springer Heidelberg, 2005.
- [167] Sebastian Thrun and Michael Montemerlo. The GraphSLAM algorithm with applications to large-scale mapping of urban structures. In *Intl. J. of Robotics Research*. Citeseer, 2006.
- [168] Sivan Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. of Research and Development*, 41(6):711–725, 1997.
- [169] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment – a modern synthesis. In *Vision Algorithms: Theory and Practice*, pages 298–372. Springer Heidelberg, 1999.
- [170] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. on Computing*, 6(3):505–517, 1977.
- [171] R. Valencia, M. Morta, J. Andrade-Cetto, and J.M. Porta. Planning reliable paths with pose SLAM. *IEEE Trans. Robotics*, 29(4):1050–1059, Aug 2013.
- [172] T. Vidal-Calleja, A.J. Davison, J. Andrade-Cetto, and D.W. Murray. Active control for single camera SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 1930–1936, May 2006.
- [173] Richard Vuduc, James W Demmel, Katherine Yelick, Shoaib Kamil, Rajesh Nishtala, Benjamin Lee, et al. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, pages 26–26. IEEE, 2002.
- [174] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In *Proc. of the USENIX Conf. on Hot Topics in Parallelism*, pages 13–13. USENIX Association, 2010.
- [175] Richard W Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proc. of the Intl. Conf. on High Performance Computing and Commun. (HPCC)*, pages 807–816. Springer Heidelberg, 2005.
- [176] Changchang Wu, Sameer Agarwal, Brian Curless, and Steven M Seitz. Multicore bundle adjustment. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 3057–3064. IEEE, 2011.

- [177] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. on Algebraic Discrete Methods*, 2(1):77–79, 1981.
- [178] Fuzhen Zhang. *The Schur complement and its applications*, volume 4. Springer US, 2005. ISBN 978-1-441-93712-4.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". It is available for \LaTeX via CTAN as `classicthesis`. A heavily modified version of this style, with numerous fixes and improvements, is available from the author's website as `classicfeces`.