BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

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 — Ing. LUKÁŠ POLOK
AUTOR PRÁCE

SUPERVISOR — Doc. RNDr. PAVEL SMRŽ, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2016 – version 1.0

CONTENTS

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. on Algebraic Discrete Methods*, 2(1):77–79, 1981.

[87] Fuzhen Zhang. *The Schur complement and its applications*, volume 4. Springer US, 2005. ISBN 978-1-441-93712-4.

[60] 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.

[61] 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.

[62] Karol Myszkowski, Oleg G Okunev, and Tosiyasu L Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–511, 1995.

[63] 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/`.

[64] J. Neira and J.D. Tard os. Data association in stochastic mapping using the joint compatibility test. *IEEE Trans. Robot. Automat.*, 17(6):890–897, December 2001.

[65] 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.

[66] NVIDIA. CUDA C programming guide version 7.5. Technical report, NVIDIA Corporation, Santa Clara, CA, 2015.

[67] Edwin Olson. *Robust and Efficient Robot Mapping*. PhD thesis, Massachusetts Institute of Technology, 2008.

[68] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search - a log log n search. *Communications of the ACM*, 21(7):550–553, 1978.

[69] 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.

[70] 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.

[71] 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. ISBN 978-981-07-3937-9.

[72] 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.

[73] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Software*, 16(4):303–324, 1990.

[74] Michael JD Powell. A hybrid method for nonlinear equations. *Numerical methods for nonlinear algebraic equations*, 7:87–114, 1970.

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

[76] Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. *ACM Journal of Experimental Algorithmics (JEA)*, 19:2–2, 2015.

[77] Richard A Snay. Reducing the profile of sparse symmetric matrices. *Bulletin Géodésique*, 50 (4):341–352, 1976.

[78] 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.

[79] Sivan Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. of Research and Development*, 41(6):711–725, 1997.

1 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 cores[1] 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 architecture that focuses on raw processing power with thousands[2] of relatively simple cores organized in three tier hierarchy, with only a very small amount of cache available (hence streaming). The memory subsystem is 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 hundreds[3] of cores based on updated Pentium designs. Although the cores in

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

[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.

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 [23] 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 [37] 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* 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 [83] and the situation can be even worse for the matrix-matrix multiplication [6, 17]. 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.

[42] 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.

[43] 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.

[44] A. Howard and N. Roy. The robotics data set repository (Radish), 2003. URL `http://radish.sourceforge.net/`.

[45] V. Ila, J. M. Porta, and J. Andrade-Cetto. Information-based compact Pose SLAM. *IEEE Trans. Robotics*, 26(1):78–93, 2010.

[46] 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. doi: 10.1109/ICRA.2015.7139841.

[47] 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.

[48] M. Kaess and F. Dellaert. Covariance recovery from a square root information matrix for data association. *Robotics and Autonomous Syst.*, 2009.

[49] 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.

[50] M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Incremental smoothing and mapping. *IEEE Trans. Robotics*, 24(6):1365–1378, Dec 2008.

[51] 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.

[52] 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.

[53] 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.

[54] Kurt Konolige. Sparse sparse bundle adjustment. In *British Machine Vision Conf. (BMVC)*, Aberystwyth, Wales, 08/2010 2010.

[55] 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.

[56] 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.

[57] 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.

[58] Joseph WH Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11(2):141–153, 1985.

[59] Kiran Kumar Matam, Siva Rama Krishna Bharadwaj, and Kishore Kothapalli. Sparse matrix matrix multiplication on hybrid CPU+GPU platforms. In *Proc. of the IEEE Intl. Conf. on High Performance Computing, Data, and Analytics (HiPC)*, 2012.

[20] Timothy A. Davis and William W. Hager. Modifying a sparse cholesky factorization, 1997.

[21] A.J. Davison and D.W. Murray. Simulataneous localization and map-building using active vision. *IEEE Trans. Pattern Anal. Machine Intell.*, 24(7):865–880, Jul 2002.

[22] 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.

[23] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.

[24] 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.

[25] Iain S Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Software*, 7(3):315–330, 1981.

[26] Iain S Duff, Roger G Grimes, and John G Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15(1):1–14, 1989.

[27] 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.

[28] C. Engels, H. Stewénius, and D. Nistér. Bundle adjustment rules. In *Symposium on Photogrammetric Computer Vision*, pages 266–271, Sep 2006.

[29] 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.

[30] 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.

[31] 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/`.

[32] Andreas Geiger, Julius Ziegler, and Christoph Stiller. StereoScan: Dense 3D reconstruction in real-time. In *IEEE Intelligent Vehicles Symp. (IV)*, 2011.

[33] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The KITTI dataset. *Intl. J. of Robotics Research*, 2013.

[34] Alan George. Nested dissection of a regular finite element mesh. *SIAM J. on Numerical Anal.*, 10(2):345–363, 1973.

[35] 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.

[36] 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.

[37] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Software*, 35(1):4, 2008.

[38] 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.

[39] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[40] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Software*, 4(3):250–269, 1978.

[41] A. Handa, M. Chli, H. Strasdat, and A. J Davison. Scalable active matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2010.

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.

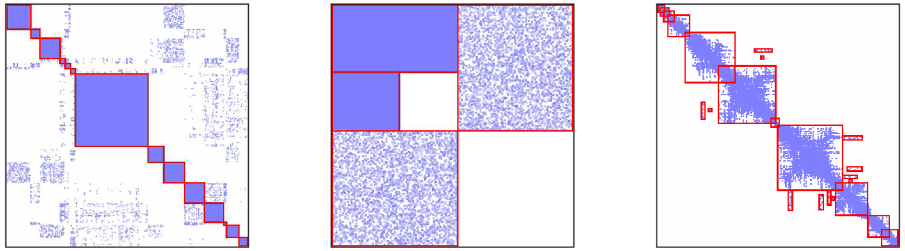
Examples of such problems can be found in robotics and computer vision. Simultaneous Localization and Mapping (SLAM) estimates the pose of a robot in conjunction with the map of the environment from various sensor measurements. Similarly, Bundle Adjustment (BA) or Structure from Motion (SfM) in computer 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 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.



(a) (b) (c)

Figure 1.1: Examples of approximately block matrices from the University of Florida Sparse Matrix Collection [18], specifically in the DIMACS10 dataset [76], a) an approximate block matrix with scattered nonzero elements, b) a block matrix with unaligned blocks, and in the Oberwolfach dataset [57], 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.

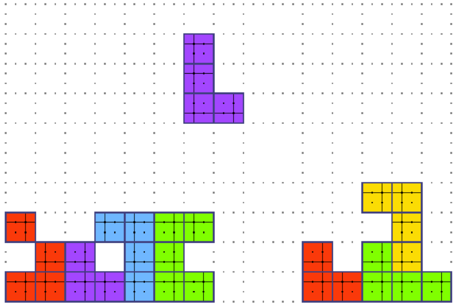


Figure 1.2: An example of a randomly generated sparse block matrix composed of 31 blocks, 3 × 3 elements each, used in testing operations on block matrices.

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 [73, 78], 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 [18]. To generate it, the algorithm from [75] 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

BIBLIOGRAPHY

[1] S. Agarwal and K. Mierle. Ceres solver. `http://ceres-solver.org/`, 2012.

[2] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001. ISBN 978-0-201-70431-0.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] A. Björck. *Numerical methods for least squares problems*. SIAM, 1996. ISBN 978-0-898-71360-2.

[8] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graphics*, 22(3):917–924, 2003.

[9] 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.

[10] 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.

[11] S. Carney, M.A. Heroux, G. Li, and K. Wu. A revised proposal for a sparse BLAS toolkit. Technical report, SPARKER Working Note, 1994.

[12] 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.

[13] Philippe Michel Cassereau. *A new class of optimal unitary transforms for image processing*. PhD thesis, MIT, 1985.

[14] 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.

[15] 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.

[16] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In *Sparse Matrices and Their Applications*, pages 157–166. Springer US, 1972.

[17] Steven Dalton, Nathan Bell, and Luke Olson. Optimizing sparse matrix-matrix multiplication for the GPU. Technical report, Technical Report, 2013.

[18] T.A. Davis. The university of florida sparse matrix collection. In *NA Digest*. Citeseer, 1994.

[19] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. SIAM, 2006. ISBN 0-89871-613-6.

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. 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.

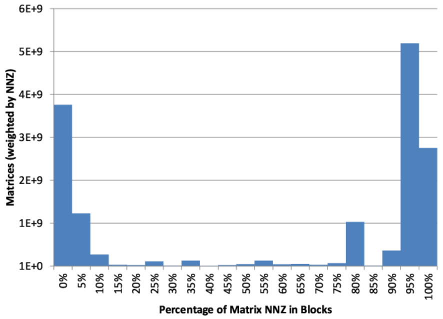


Figure 1.3: Distribution of data between elementwise and block sparse matrices in the University of Florida Sparse Matrix Collection [18].

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, 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 [24, 25, 73, 47, 83, 85] to consolidate general sparse matrices into block matrices, making acceleration of problems without inherent block structure also possible.

2 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.

Block matrices can be more or less permissive as to the shape and placement of the dense blocks. 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 2.1. The only downside is that in some cases, the fragmentation can leave many 1×1 blocks behind or even yield an elementwise sparse matrix.

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 2.1a. Unaligned block matrices (Figure 1.1b or Figure 2.1c) arise naturally e.g. in LOTs [12, 13] in image processing, where each two adjacent blocks overlap in order to avoid discontinuities in the processed image.

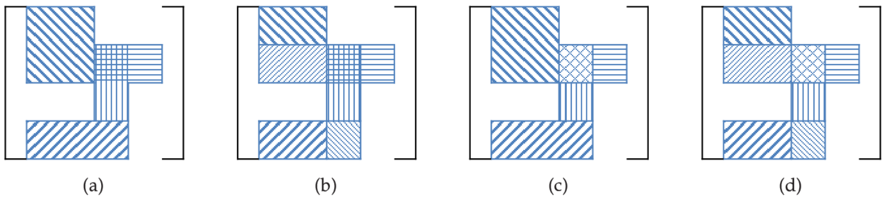


Figure 2.1: Block placements in sparse block matrices: a) unaligned block matrix with four blocks, two of which overlap, b) aligned block matrix – the unaligned blocks were fragmented (now there are 8 blocks two of which still overlap), c) unaligned with the overlapping blocks fragmented and fused (total of 5 blocks) and d) aligned non-overlapping block matrix (7 blocks).

10 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.
- Efficient Arithmetics for Sparse Block Matrices.
- Sparse Block Matrix Factorizations.
- Efficient Variable Reordering Strategy for Incremental Cholesky Factorization.
- Analysis of the Computational Complexities in Schur Complement.
- Clique-Based Ordering for Schur Complement.
- Incremental Schur Complement.
- Sparse Block Matrix Formulation of the Recursive Formula.
- Incremental Covariance Matrix Update and Downdate.
- Sparse Covariance Recovery for Schur Complemented Systems.
- Fast GPU Sorting Kernel.
- Fast GPU Sparse Matrix Multiplication Kernel.

10.1 FUTURE WORK

The sparse block matrix factorizations presented here, despite being highly competitive and outperforming even state of the art implementations such as Cholmod [20], 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 [40]), 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 simplical. Their supernodal forms can be implemented to gain significantly better performance. Efficient multifrontal or parallel CPU implementations would also yield a considerable speedup.

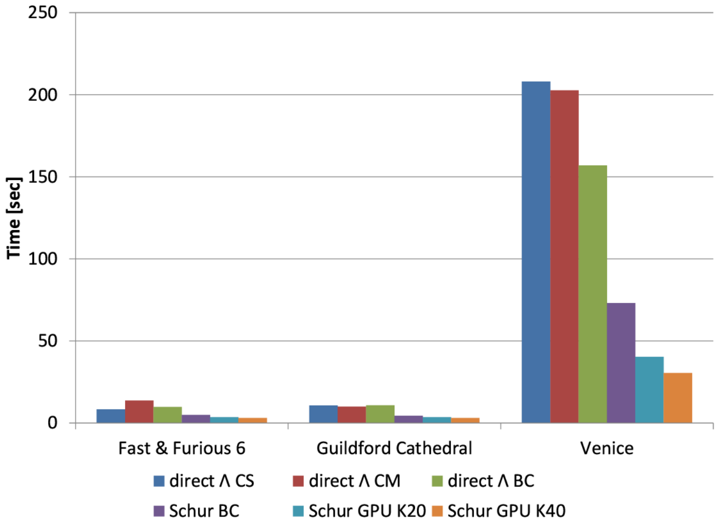


Figure 9.1: GPU-accelerated NLS solving performance on the standard BA datasets.

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 an 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 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.

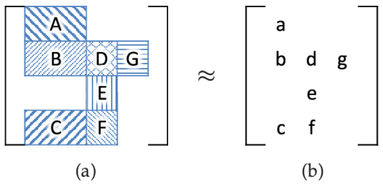


Figure 2.2: Relation of expressions on block and elementwise sparse matrices: a) aligned non-overlapping block matrix and b) a structurally equivalent elementwise sparse matrix.

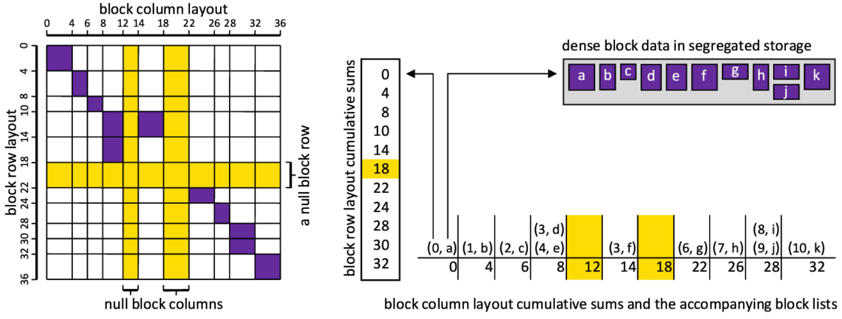
Assuming aligned, non-overlapping matrices has its benefits. Each block of the matrix can be treated as a (scalar) variable in an ordinary (elementwise) sparse matrix and formulas applicable to the elements can be automatically extended to blocks (see Figure 2.2), with the difference that scalar operations 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.

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 [19] or Cholmod [20] 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 [75] 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.


block column layout
0 4 6 8 12 14 18 22 26 28 32 36
block row layout
0 4 8 10 14 18 22 24 28 30 32 36
a null block row
null block columns
dense block data in segregated storage
a b c d e f g h i j k
block row layout cumulative sums
0 4 8 10 14 18 22 24 28 30 32
(0, a) (1, b) (2, c) (3, d) (4, e) (3, f) (6, g) (7, h) (8, i) (9, j) (10, k)
0 4 6 8 12 14 18 22 26 28 32
block column layout cumulative sums and the accompanying block lists

(a)

(b)

Figure 2.3: 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.

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 [75] 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 blockwise 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.

2.1 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.

9 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 Canaria[1]. *Guildford Cathedral* comprises of 92 DSLR stills of the Guildford Cathedral[2] (Surrey, London). *Venice* is a standard bundle adjustment dataset [55] 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 CPUs 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 9.1. The solutions using a direct solver without the Schur complement are denoted direct-Λ-CS (CSparse [19]), direct-Λ-CM (Cholmod [20]) and direct-Λ-BC (the block Cholesky proposed in Chapter 2). 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 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.

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

[1] Kindly provided by Double Negative Visual Effects, http://www.dneg.com/.

[2] Freely available at http://cvssp.org/impart/, upon request.

estimate the saturated costs to 27.7 ms/MFLOP for CSparse, 4.2 ms/MFLOP for CUSP and finally 3.0 ms/MFLOP for the proposed.

A more conventional comparison is presented in Table 8.1. This comparison was performed on the SNAP subset of the University of Florida Sparse Matrix Collection. It contains 9 different classes of matrices, a single matrix was chosen from each of them, much like in the evaluation in [59]. 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 (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 8.3b.

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.

In the g2o [55] 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 [1] implementation, the blocks are allocated in a linear array which would necessitate reallocation and data copying when incrementally adding new 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 element-wise 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.

2.1.1 *The Data Structure*

In general, a vast majority of the existing block matrix schemes, including the proposed one, involves the same data layout as the 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 [1, 54, 55], 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 2.3a on the top and left edge of the matrix. In addition, rhe columns structure contains the lists of non-zero matrix blocks, each comprising of a row index and a pointer to matrix data.

The elements themselves are stored in forward-allocated segregated storage (see Figure 2.3b), 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 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 Streaming SIMD Extensions (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, 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 2.3a and their representation is shown in Figure 2.3b 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.

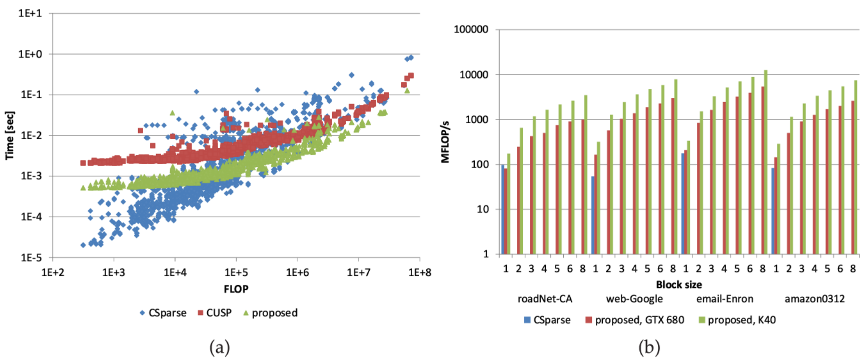
This contrasts with the usual sparse block matrix representations, which only describe the layout of nonzero blocks without caring about the null elements in between. It comes at the cost of small increase in memory requirements, but only for the layout itself, not for the data. If n_b and m_b are the 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. lease 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.

2.1.2 *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.

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



(a)

(b)

Figure 8.3: a) Performance scaling comparison on Tesla K40. Note that both axes are logarithmic, b) Performance scaling comparison of sparse block matrix multiplication on the first four matrices from the SNAP dataset.

Table 8.1: 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 CSparse and CUSP.

	GF GTX 680		Tesla K40				
Matrix	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	*out of RAM*		0.497	**0.446**	214.127	30.089	1.114
as-Skitter	*out of RAM*[5]						

not include the conversion or data transfers. The benchmarked version of the proposed algorithm handles all the rank deficient cases. The memory for the expansion and the product was allocated online, 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 8.3a. 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 [40]. The times of the parallelized implementations grow slowly before the GPU gets saturated, then also scale approximately linearly. Least squares were employed to

[5] Note that with only 16 GB of RAM, this matrix is too large even for CSparse: the product would take 26.4 GB.

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.

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. However, radix sort is not a good candidate for segmented implementation, as it would lead to both runtime and space tolls. Fortunately, for merge sort, segmented variants exist[2], 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 8.1a where sorting takes only 34%, with Figure 4 in [17] where it is closer to 63% of the time.

8.1.3 *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 8.2b bottom). A simple segmented reduction can be 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.

8.2 RESULTS

In this section, the timing results of sparse matrix multiplication performed using the proposed implementation[3] are compared with a similar state of the art implementation, CUSP 0.3.1 [63]. It was also compared to CSparse 1.2.0 [19], which runs on the CPU[4]. 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 [18] and their transposes (for matrices which share a common dimension).

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. 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

[2] One such implementation can be found at `http://nvlabs.github.io/moderngpu/segsort.html`.

[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.

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.

2.1.3 *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. Other existing block matrix implementations require $O(\log n_b)$ lookup.

To improve performance, a function, mapping block rows of **B** to block columns of **A** can be used. Consider Algorithm 2.1: 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. 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. 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. Also 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.

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 $A^\top A$ form.

As mentioned above, the block sizes correspond to the DOF of the variables and, in general, are known in advance. Using typelists [2] and templates, decision trees 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 2.1 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.

2.1.4 *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

Algorithm 2.1: Fast sparse block matrix multiplication.

```
function FastMult($\mathbf{A}$, $\mathbf{B}$)
    $\mathbf{C} = \textsc{NewMatrix}(\textsc{Rows}(\mathbf{A}), \textsc{Cols}(\mathbf{B}))$
    $\mathbf{f}_{\text{map}} = \textsc{BlockLayoutMapping}(\textsc{BlockCols}(\mathbf{A}), \textsc{BlockRows}(\mathbf{B}))$
    $\text{colB}_{\text{id}} = 0$
    for each $\text{columnB}_{\text{block}}$ in $\mathbf{B}$ do
        for each blockB in $\text{columnB}_{\text{block}}$ do
            $\text{rowB}_{\text{id}} = \textsc{RowIdOf}(\text{blockB})$
            $\text{columnA}_{\text{id}} = \mathbf{f}_{\text{map}}(\text{rowB}_{\text{id}})$
            if $\text{columnA}_{\text{id}} = \textit{mismatch}$ then
                return                    ▷ block layout mismatch, product not defined
            else if $\text{columnA}_{\text{id}} = \textit{null}$ then
                continue               ▷ the column in $\mathbf{A}$ is mismatched but also empty
            end if
            $\text{columnA}_{\text{block}} = \textsc{BlockCols}(\mathbf{A})[\text{columnA}_{\text{id}}]$        ▷ $O(1)$
            for each blockA in $\text{columnA}_{\text{block}}$ do
                $\text{rowA}_{\text{id}} = \textsc{RowIdOf}(\text{blockA})$
                $\text{block}_{\text{dest}} = \textsc{FindBlockLog}(\text{rowA}_{\text{id}}, \text{colB}_{\text{id}}, \mathbf{C})$        ▷ $\leq O(\log f_b)$
                $\text{block}_{\text{dest}} = \text{block}_{\text{dest}} + \text{blockA} \cdot \text{blockB}$
            end for
        end for
        $\text{colB}_{\text{id}}{+}{+}$
    end for
    return $\mathbf{C}$
end function
```

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:

$$\begin{pmatrix} \epsilon & 1 \\ \mathbf{1} & 0 \end{pmatrix}. \tag{2.1}$$

Eliminating the $\mathbf{1}$ 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.

8.1.1 *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 [40], it is trivial to derive the exact size of the expansion:

$$\text{expansion}(A, B) = \sum_{j=1}^{\text{cols}(B)} \sum_{k=1}^{\text{nnzc}(B,j)} \text{nnzc}(A, \text{row}(B, j, k)), \tag{8.1}$$

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 [6, 17]). Figure 8.1b plots a ratio of expansion size to the number of nonzeros in the product. In certain cases 100× more storage than the final product is required (please, refer to Section 8.2 for the description of the dataset). 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.

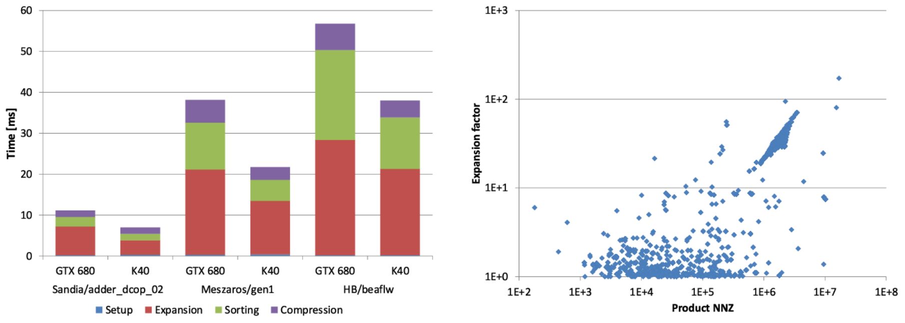
The choice of granularity of expansion is crucial to load balancing. The proposed algorithm achieves perfect load balancing in the expansion stage by using the granularity of individual scalar products. To do that, it is necessary for each thread to find the elements of A and B to process. Here, the *interpolation search* [68] 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 expanded scalar products are essentially the product matrix in the `COO` format; they can be stored in three vectors of the same length, $\boldsymbol{ex}_{\text{cols}}$ contains column indices, $\boldsymbol{ex}_{\text{rows}}$ contains row indices and $\boldsymbol{ex}_{\text{values}}$ contains values of the elements (see Figure 8.2b). Note that the sparse multiplication algorithm generates a partially ordered expansion, where $\boldsymbol{ex}_{\text{cols}}$ is ordered and $\boldsymbol{ex}_{\text{rows}}$ consists of many short ordered runs (given by the rows of elements in the columns of A, which are typically ordered).

8.1.2 *Sorting Stage*

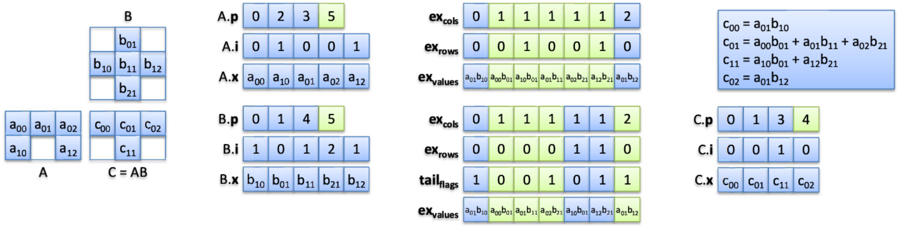
The approach in [6] is to use a single global sort. On GPU, the most efficient sort implementations use radix sort [60] with complexity $O(kN)$ where k is proportional to the number of bits of the key. In the case of keys generated by the expansion, the number of bits is given by base 2 logarithm of 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



(a) (b)

Figure 8.1: a) Time of different stages of the proposed algorithm, b) Expansion factor by the number of product nonzero entries.


B
b_{01}
b_{10} b_{11} b_{12}
b_{21}
a_{00} a_{01} a_{02}
a_{10} a_{12}
A
c_{00} c_{01} c_{02}
c_{11}
C = AB
A.p 0 2 3 5
A.i 0 1 0 0 1
A.x a_{00} a_{10} a_{01} a_{02} a_{12}
B.p 0 1 4 5
B.i 1 0 1 2 1
B.x b_{10} b_{01} b_{11} b_{21} b_{12}
ex_{cols} 0 1 1 1 1 1 2
ex_{rows} 0 0 1 0 0 1 0
ex_{values} $a_{01}b_{10}$ $a_{00}b_{01}$ $a_{10}b_{01}$ $a_{01}b_{11}$ $a_{02}b_{21}$ $a_{12}b_{21}$ $a_{01}b_{12}$
ex_{cols} 0 1 1 1 1 1 2
ex_{rows} 0 0 0 0 1 1 0
$tail_{flags}$ 1 0 0 1 0 1 1
ex_{values} $a_{01}b_{10}$ $a_{00}b_{01}$ $a_{01}b_{11}$ $a_{02}b_{21}$ $a_{10}b_{01}$ $a_{12}b_{21}$ $a_{01}b_{12}$
$c_{00} = a_{01}b_{10}$
$c_{01} = a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21}$
$c_{11} = a_{10}b_{01} + a_{12}b_{21}$
$c_{02} = a_{01}b_{12}$
C.p 0 1 3 4
C.i 0 0 1 0
C.x c_{00} c_{01} c_{11} c_{02}

(a) (b) (c) (d)

Figure 8.2: Data at the individual stages of the ESC algorithm[1], 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.

8.1 ALGORITHM DESIGN

The algorithm introduced in this chapter is based on the ESC algorithm [6, 17]. However, the focus is on removing load imbalances and on simplicity, as especially the improved ESC algorithm in [17] 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.

[1] An interactive demonstrator is available online at `http://www.fit.vutbr.cz/~ipolok/esc`.

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 product[1] $R^\top R$. 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 [39] 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). 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 left-looking column algorithm is taken as the starting point. Implementing it is trivial – square root becomes dense Cholesky decomposition, division becomes back-substitution with multiple right hand sides. The only tricky operation is a dot product of two columns becomes $\Lambda_{i,k}^\top \cdot \Lambda_{i,j}$ which 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 [19], 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 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 *simplical*, Cholmod implements a supernodal factorization [14] 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 the Cholesky factorization 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 can-

[1] 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.

cellation). 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:

$$\Lambda = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 0 \\ 1 & 0 & 2 \end{pmatrix}, P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, P^\top \Lambda P = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \quad (2.2)$$

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 $P^\top \Lambda P$. 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 $\mathbf{p} = (2, 0, 1)$).

Finding the best fill-reducing permutation is an NP-complete problem [86], 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, [36, 77, 30, 27], notably Reverse Cuthill-McKee (RCM) [16]. Later, orderings based on the elimination graph were proposed such as Exact Minimum Degree (EMD) and its modifications [58], Approximate Minimum Degree (AMD) [4] or Nested Dissection [34].

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 [52].

2.2 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, and the version with metaprogramming optimization is denoted UBlock FBS (fixed block size).

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. The evaluation was performed on a subset of the The University of Florida Sparse Matrix Collection [18]. This collection was chosen because it contains sparse matrices corresponding to a diverse set of problems, and as

8 FAST SPARSE MATRIX MULTIPLICATION ON GPU

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 [87].

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`) [19] 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 [40] is implemented e.g. in the popular CSparse package [19] (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 not canonical.

The algorithm [40] 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 [40] came up with an elegant way of quickly merging these partially ordered sequences.

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 [5, 31] use this approach.

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 [62] or matrix multiplication using multi-texturing and blending functions [56]. Addition of programmable shaders allowed implementation of more complex algorithms, e.g. sparse matrix solvers [8].

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. [66], or other plentiful material available on this topic.

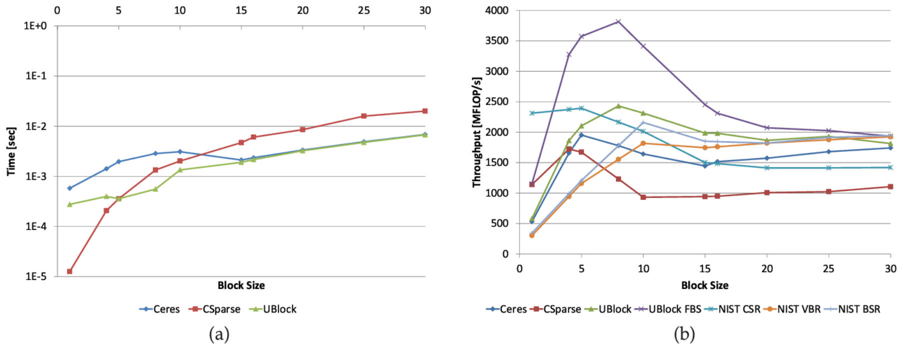


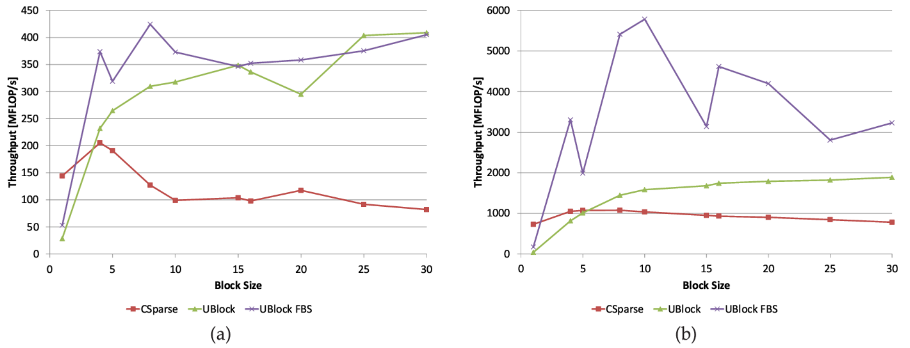
Figure 2.4: a) Time for compression of the MCCA matrix (smaller is better), b) Performance scaling of general matrix vector product on the MCCA matrix.

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 [26] 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 experimental evaluation [11] on it. Since the NIST BLAS is not widely used, this limited comparison should be sufficient.

A comparison of the time required to compress a sparse matrix using CSparse, Ceres and our implementation is shown in Figure 2.4a. 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 2.4b 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



(a)

(b)

Figure 2.5: a) Performance scaling of linear combination of the MCCA matrix and its transpose, b) Performance scaling of the product of the MCCA matrix and its transpose.

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 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 2.5a. 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 2.5b 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 Cachegrind[2] 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%.

[2] A part of the Valgrind tool family, see `http://www.valgrind.org/info/tools.html#cachegrind`.

7 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 5 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 [84]. 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.

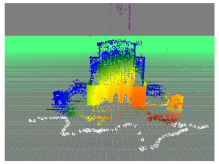
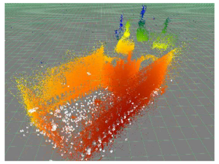
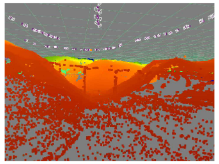
BLOCK DIAGONAL MATRIX INVERSE: used in Schur complement, generally not a bottleneck (except if using AMICS ordering where the diagonal blocks can get large) but is easily parallelizable.

7.1 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 FPS 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.

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

Table 6.2: Characteristics of the BA datasets used in covariance recovery evaluations.

	Cathedral	Venice	Fast & Furious 6
			
Cameras	92	871	160
Landmarks	57, 957	530, 304	136, 453
Visibility	7.28 obs. / lm.	5.35 obs. / lm.	3.42 obs. / lm.
Λ	142.93 MB	980.33 MB	167.70 MB
Schur(A)	1.04 MB	45.06 MB	1.14 MB
S	1.04 MB	84.60 MB	1.73 MB

which was kindly provided by Double Negative Visual Effects[5]. Two additional methods were compared: recursive formula on Cholesky factor of the system matrix, and recursive formula on Schur(D) as in (6.11). More details about the datasets are listed in Table 6.2.

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 [70] were used throughout the whole implementation, which previously proved about an order of magnitude speedups for batch recursive formula [46]. Block matrix products and decompositions were accelerated by Tesla K20x GPU.

Times required to calculate the marginal covariances are reported in Table 6.1. 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.

The magnitudes of the calculated landmark covariances are displayed as false color, see Figure 6.2 or Table 6.2. From the colored view, it is apparent which parts of the reconstruction are more precise and which are not. The user can take advantage of such images to re-capture poorly reconstructed areas and obtain a high accuracy 3D reconstruction.

[5] http://www.dneg.com/

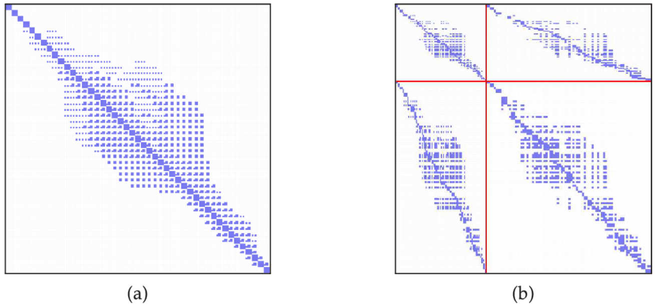


Figure 2.6: The MCCA matrix a) with the elements inflated to blocks of 4 different sizes and b) its split form.

In solving FEM problems and perhaps also in other methods which rely on highly efficient matrix vector products, an approach called *splitting* [79, 69, 35] 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 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 Floating Point Operations (FLOPs) would be the same for all the tests. An example for four different block sizes is given in Figure 2.6. 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 2.7. 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. 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

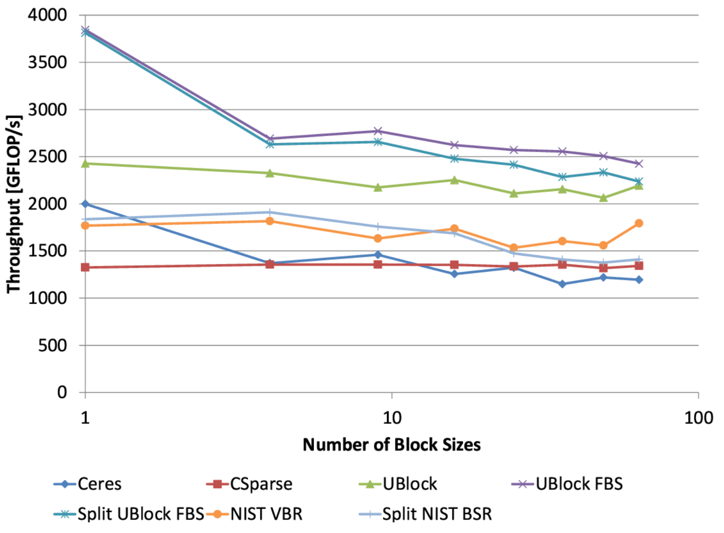


Figure 2.7: Comparison of splitting and the variable block size approaches.

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.

Table 6.1: Timing results and the associated space requirements of the evaluated covariance recovery methods (best times in bold).

Dataset	Cathedral	Venice	Fast & Furious 6
Landmarks, using (6.14)	**0.165** s	**7.060** s	**0.293** s
Size of sparse $S^{-\top}$	1.24 MB	109.79 MB	2.97 MB
$\mathrm{Chol}(\Lambda)$	1.251 s	16.856 s	0.951 s
Rec. formula all	3.308 s	82.689 s	3.493 s
Size of $\mathrm{Chol}(\Lambda)$	74.05 MB	572.52 MB	93.33 MB
$\mathrm{Schur}(D)$	160.662 s	4457.539† s	149.999 s
$\mathrm{Chol}(\mathrm{Schur}(D))$	73† hours	N/A	139† hours
Rec. formula lm.	4459.647† s	N/A	5† hours
Size of $\mathrm{Schur}(D)$	37.87 GB	398.01 GB	46.95 GB
Size of $\mathrm{Chol}(T)$	106.70 GB	~ 7.37 TB	493.43 GB
Cameras, using (6.10)	0.028 s	38.809 s	0.045 s

is no longer block diagonal and inverting it is much more difficult than inverting D in (5.2). Applying the Woodbury formula to (6.11) gives:

$$\boldsymbol{\Sigma}_l = D^{-1} + D^{-1}U^\top(A - UD^{-1}U^\top)^{-1}UD^{-1}\,, \tag{6.12}$$

$$\boldsymbol{\Sigma}_l = D^{-1} + D^{-1}U^\top\boldsymbol{\Sigma}_p UD^{-1}\,, \tag{6.13}$$

$$\boldsymbol{\Sigma}_l = D^{-1} + D^{-1}U^\top S^{-1}S^{-\top}UD^{-1}\,. \tag{6.14}$$

Evaluating all of (6.14) would yield a dense matrix with the size approaching that of the full $\boldsymbol{\Sigma}$ which would be counterproductive. Instead, taking advantage of symmetry of Λ (and thus also of D and $\boldsymbol{\Sigma}$), it is possible to write $\mathbf{B} \triangleq S^{-\top}UD^{-1}$ in order to get $\boldsymbol{\Sigma}_{l_{i,j}} = \mathbf{D}^{-1}_{i,j} + \mathbf{B}^\top_{i,*} \cdot \mathbf{B}_{*,j}$ where $\mathbf{D}$ 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; $S^{-\top}$ 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 (5.3) with covariance in place of $\boldsymbol{\alpha}$ and identity on the right:

$$\boldsymbol{\Sigma}_{pl} = (S^\top S)\backslash(0 - UD^{-1}I_l)\,, \tag{6.15}$$

$$\boldsymbol{\Sigma}_{pl} = -\boldsymbol{\Sigma}_p UD^{-1}\,. \tag{6.16}$$

Again, computation can be saved by taking advantage of sparsity of the matrices so that recovering the full $\boldsymbol{\Sigma}_p$ is not necessary.

6.2.1 *Experimental Evaluation*

The proposed method for recovering marginal covariances of points was tested on two public datasets, the *Guildford Cathedral*[4], *Venice* [55] and on the *Fast & Furious 6* dataset

[4] can be obtained at `http://cvssp.org/impart/`

The performance of our incremental NLS solver in [71] 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.

6.2 RECOVERING COVARIANCE IN SCHUR COMPLEMENTED SYSTEMS

In Section 6.1, 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 5, some of the problems can be solved more efficiently using Schur complement rather than by directly factorizing the system matrix A or Λ. In those cases, it is still possible to e.g. use the recursive formula (6.1) and (6.2) 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 $\Lambda\boldsymbol{\Sigma} = \mathrm{I}$ directly on the Schur complemented system:

$$\begin{pmatrix} A & U \\ U^\top & D \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{\Sigma}_p & \boldsymbol{\Sigma}_{pl} \\ \boldsymbol{\Sigma}_{pl}^\top & \boldsymbol{\Sigma}_l \end{pmatrix} = \begin{pmatrix} I_p & 0 \\ 0^\top & I_l \end{pmatrix} . \qquad (6.9)$$

where both $\boldsymbol{\Sigma}$ and the identity matrix I are partitioned the same way as Λ is partitioned in (5.2). Note that the subscripts here are only identifiers rather than element indices. By taking Cholesky decomposition $S^\top S \triangleq \mathrm{Schur}(A)$, the covariances of the camera variables are:

$$S^\top S \boldsymbol{\Sigma}_p = I_p - UD^{-1}0^\top = I_p \quad \text{so} \quad \boldsymbol{\Sigma}_p = (S^\top S)^{-1} , \qquad (6.10)$$

and thus the recursive formula in (6.1) and (6.2) can be used efficiently.

The situation is more interesting in recovering the covariances of the landmarks. It would be possible to make use of $T^\top T \triangleq \mathrm{Schur}(D)$ and (6.10) to write:

$$\boldsymbol{\Sigma}_l = (D - U^\top A^{-1} U)^{-1} = (T^\top T)^{-1} . \qquad (6.11)$$

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

3 BATCH SOLVING IN SLAM ++

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 [28] or elastodynamic simulations in computer graphics [43] 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. 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 [22, 38, 50, 51, 55]. Graphs allow more natural representation of nonlinear 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 Degree of Freedom of the variables.

Some of the existing implementations rely on sparse block-structure schemes [54, 55]. The block structure is maintained until the point of solving the linear system. Here is where CSparse [19] or Cholmod [20] libraries are used to perform the matrix factorization. Those are state of the art elementwise implementation of operations on sparse matrices.

3.1 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 the *normal equation* 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 [22]:

$$\hat{A} = \begin{pmatrix} A \\ A_u \end{pmatrix}, \hat{\mathfrak{b}} = \begin{pmatrix} \mathfrak{b}_1 \\ \mathfrak{b}_2 + \mathfrak{b}_u \end{pmatrix}, \tag{3.1}$$

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

$$J_k = \left(0 \ldots \frac{\partial r_k}{\partial \theta_{i_k}} \ldots 0 \ldots \frac{\partial r_k}{\partial \theta_{j_k}}\right). \tag{3.2}$$

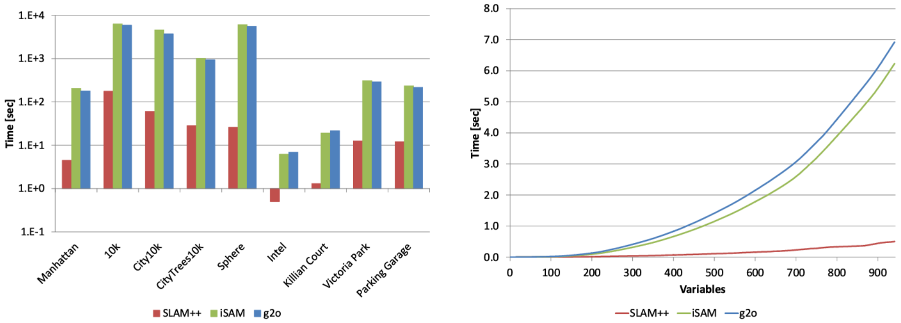
Note that the additions in (3.1) may require padding A with new zero columns and $\mathfrak{b}_2$ with new zero rows in case new variables are added. Extension to multiple new measurements is trivial.

Similarly, for the Λ matrix in the normal equation, the increments translate to adding new block rows and block columns (as Λ is symmetric) with the size of each new variable. Updates translate to (potentially) adding new nonzero entries. Updating Λ and η is additive:

$$\hat{\Lambda} = \begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^\top & \Lambda_{22} + \Omega \end{pmatrix}, \hat{\eta} = \begin{pmatrix} \eta_1 \\ \eta_2 + \boldsymbol{\omega} \end{pmatrix}, \tag{3.3}$$

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

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 [50, 51, 52] proposed efficient algorithms to incrementally solve the linear systems. Those



(a) (b)

Figure 6.5: Covariance recovery performance evaluation; a) logarithmic plot of time on standard datasets and b) cumulative time on the *Intel* dataset.

6.1.2 *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 [50] and g2o [55]. Both, iSAM and g2o use fairly similar implementation of the recursive formula (6.1), (6.2) together with a cache of already calculated 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 ++ library[3]. 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 $\boldsymbol{\Sigma}$ and to have explicit A_u or Ω.

The evaluation was performed on the standard robotics datasets. The tests were performed on a computer with Intel Core i5 CPU 661 running at 3.33 GHz and 8 GB of RAM.

6.1.2.1 *Time evaluation*

Figure 6.5a reports the covariance recovery time on logarithmic scale while Figure 6.5b shows the cumulative time of the incremental covariance computation on the *Intel* dataset during the execution of the algorithm. An approximate time complexity 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 $O(n^{2.36})$.

[3] http://sf.net/p/slam-plus-plus/

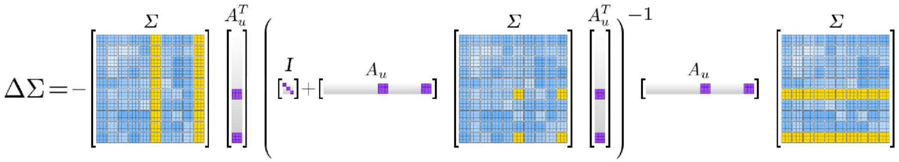


Figure 6.4: Sparsity patterns involved in covariance update calculation.

measurements involved in the update. For the simple case of a single measurement of a given DOF, m_u = 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 (6.5) becomes a simple block vector multiplication:

$$\triangle\boldsymbol{\Sigma} \quad = \quad -\mathbf{B}S^{-1}\mathbf{B}^\top \quad \text{where} \quad \mathbf{B} \quad = \quad \boldsymbol{\Sigma}A_u^\top \quad \text{is a block vector.} \tag{6.6}$$

Due to the sparsity of the A_u, only few elements of the full $\boldsymbol{\Sigma}$ 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 6.4. Furthermore, the size of $\mathbf{B}$ is $n \times m_u$, but the product in (6.6) is a full matrix. Therefore the computation of the entire $\triangle\boldsymbol{\Sigma}$ is prohibitive. We mentioned above that only some elements of the covariance are needed in the applications. For a single block, $\hat{\boldsymbol{\Sigma}}_{i,j}$, the update can be easily calculated as:

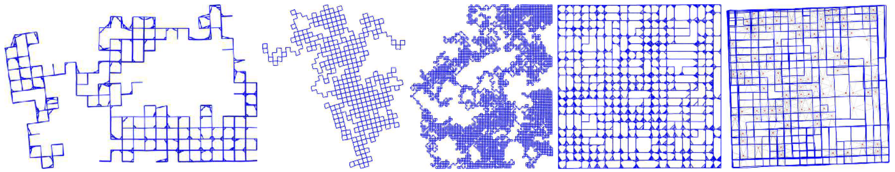
$$\triangle\boldsymbol{\Sigma}_{i,j} \quad = \quad -\mathbf{B}_i S^{-1}\mathbf{B}_j^\top\,, \tag{6.7}$$

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 [45] 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 $\boldsymbol{\Sigma}$ must be avoided. Only the blocks required by the application (for instance only the diagonal of $\boldsymbol{\Sigma}$) are stored in a sparse block matrix. However, in order to compute the update in (6.6) or in (6.7), other elements of $\boldsymbol{\Sigma}$ are needed (the block columns, corresponding to the variables $\boldsymbol{\nu}$, involved in the update). Those are obtained by solving the system:

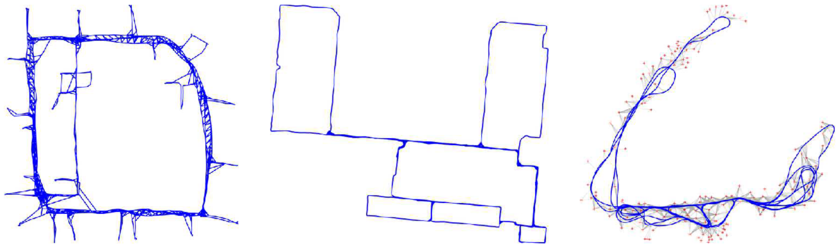
$$\Lambda\boldsymbol{\Sigma}_{*,\nu} \quad = \quad \mathbf{I}_{*,\nu} \quad \text{or} \quad \mathbf{R}\boldsymbol{\Sigma}_{*,\nu} \quad = \quad \mathbf{R}^{-\top}\mathbf{I}_{*,\nu}\,, \tag{6.8}$$

where $\mathbf{I}$ is an identity matrix of the same size as $\mathbf{R}$ and $\mathbf{I}_{*,\nu}$ 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 $\boldsymbol{\Sigma}_{*,\nu}$ by forward and back substitution amounts to $O(2n_{nz})$.



(a) Manhattan (b) 10k (c) 100k (d) City10k (e) CityTrees10k

Figure 3.1: The synthetic datasets used in the batch solver evaluations.



(a) Intel (b) Killian (c) Victoria Park

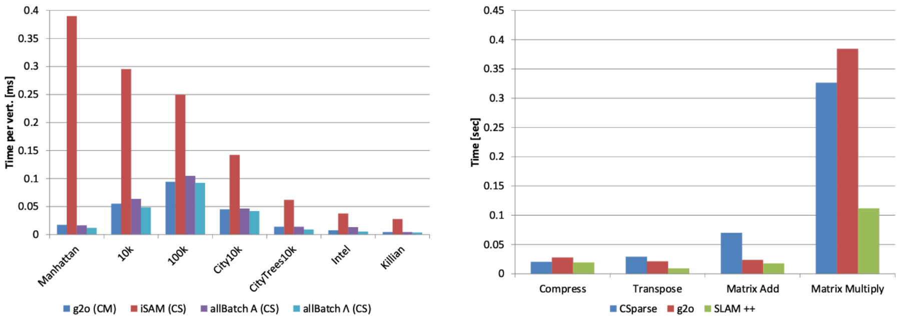
Figure 3.2: The real-world datasets used in the batch solver evaluations.

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.

3.2 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 $\mathbf{J}_k$ 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.



(a) (b)

Figure 3.3: a) Comparisons of time *per vertex* in the batch solvers (CM refers to Cholmod and CS refers to CSparse), b) 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 [72].

3.3 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 $A\mathbf{x} = \mathbf{b}$, that is denoted allBatch-A and another one that increments the information matrix in the normal equation $\Lambda\mathbf{x} = \boldsymbol{\eta}$ where $\Lambda = A^\top A, \boldsymbol{\eta} = A^\top \mathbf{b}$, that is denoted allBatch-Λ. The timing results were compared to similar state of the art implementations such as iSAM [50], g2o [55], and SPA [53] (a 2D SLAM variant of sSBA [54]).

The implementations were evaluated on five standard simulated datasets; *Manhattan* [67], *10k* and *100k* [38], *City10k* and *CityTree10k* [49] and on three real datasets; *Intel* [44], *Killian Court* [9] and *Victoria park* [65] (see Figure 3.1 and Figure 3.2). These are 2D SLAM datasets commonly used in evaluating graph-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.

3.3.1 *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 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.

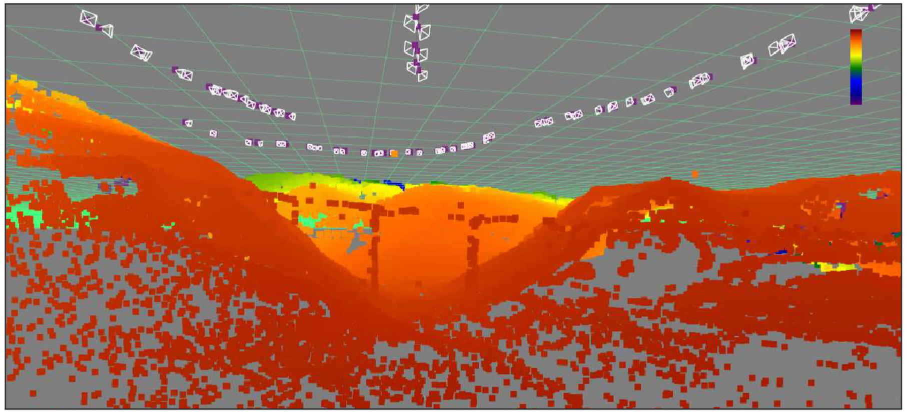


Figure 6.2: Marginal covariances used as a quality estimate of 3D reconstruction of the bridge sequence from the *Fast & Furious 6* dataset, displayed in false colors (orange means high confidence, blue – low confidence). Data courtesy of Double Negative Visual Effects.

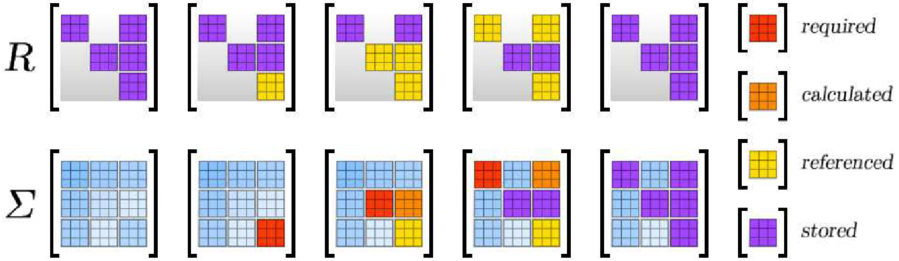


Figure 6.3: Recovering the diagonal of the covariance matrix using the recursive formula.

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

$$\begin{aligned}\hat{\Sigma} &= \Lambda^{-1} - \Lambda^{-1} A_u^\top (I + A_u \Lambda^{-1} A_u^\top)^{-1} A_u \Lambda^{-1}\,, \\ \hat{\Sigma} &= \Sigma - \Sigma A_u^\top (I + A_u \Sigma A_u^\top)^{-1} A_u \Sigma\,. \end{aligned} \tag{6.4}$$

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

$$\hat{\Sigma} = \Sigma + \triangle\Sigma\,, \; \triangle\Sigma = -\Sigma A_u^\top (I + A_u \Sigma A_u^\top)^{-1} A_u \Sigma\,. \tag{6.5}$$

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 precisely[2], the size of S, $m_u \times m_u$ with $m_u \ll m$, corresponds to the

[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.

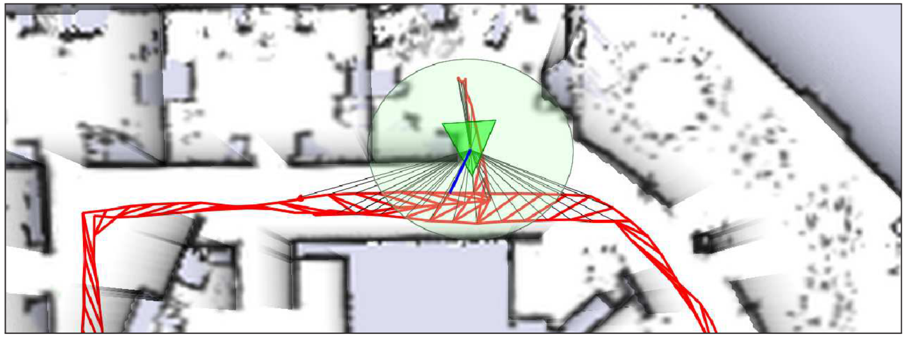


Figure 6.1: Distance-based candidates for data association calculated using the marginal covariances (95% confidence interval shown in green), on the *Intel* dataset.

block elements. In [48], it was shown how specific elements from the covariance matrix can be efficiently calculated from the $\mathbf{R}$ factor by applying the recursive formula:

$$\Sigma_{i,i} = \frac{1}{\mathbf{R}_{i,i}} \left(\frac{1}{\mathbf{R}_{i,i}} - \sum_{k=i+1, \mathbf{R}_{i,k} \neq 0}^{n} \mathbf{R}_{i,k} \Sigma_{k,i} \right), \tag{6.1}$$

$$\Sigma_{i,j} = \frac{1}{\mathbf{R}_{i,i}} \left(\sum_{k=i+1, \mathbf{R}_{i,k} \neq 0}^{j} \mathbf{R}_{i,k} \Sigma_{k,j} - \sum_{k=j+1, \mathbf{R}_{i,k} \neq 0}^{n} \mathbf{R}_{i,k} \Sigma_{j,k} \right). \tag{6.2}$$

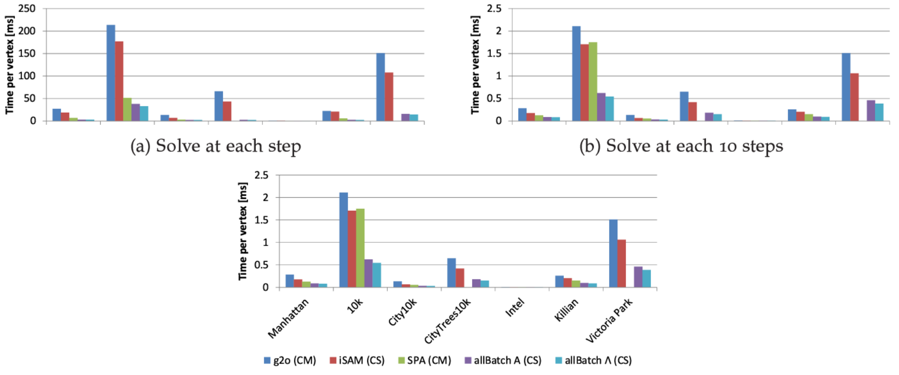
Note that above, the computations are carried out by blocks; the numerical result is the same as if computed by elements but the calculation can be performed more efficiently. In case that $\mathbf{R}$ is sparse, the formulas above can be used to compute the blocks of $\boldsymbol{\Sigma}$ at the positions of nonzero blocks in $\mathbf{R}$ quickly [7]. 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 6.3[1] shows which elements need to be calculated for a specific block diagonal element.

6.1.1 *Incremental Update of the Covariance Matrix*

In Chapter 4, 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 every new measurement can be easily integrated into the current system matrix Λ by a simple addition (see (3.3)), but things get complicated when the covariance is required:

$$\hat{\boldsymbol{\Sigma}} = (A^\top A + A_u^\top A_u)^{-1} = (\Lambda + \Omega)^{-1}. \tag{6.3}$$

[1] An insightful animation of the covariance recovery is available online at `http://slam-plus-plus.sf.net/cov/`



(a) Solve at each step

(b) Solve at each 10 steps

(c) Solve at each 100 steps

Figure 3.4: Comparisons of time *per vertex* in the batch solvers running in incremental mode (CM refers to Cholmod and CS refers to CSparse).

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 10^{th} or every 100^{th} step, respectively. iSAM, on the other hand, is able to reduce this error in every step of the algorithm even between the 10^{th} or 100^{th} 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.

3.3.2 *Discussion of the Results*

Timing results for running batch and incremental SLAM are shown in Figures 3.3a and 3.4, 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. 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 implementation 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 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.

3.3.3 *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 elementary sparse matrix operations, such as *compression*, *transpose*, *addition* and *multiplication* were measured. Performance of CSparse [19], g2o [55] and our implementation were compared. SPA [53] was not included because it's block matrix scheme is similar as in g2o. iSAM [50] was not included either, since it does not use any block matrix scheme. The results are shown in Figure 3.3b.

Observe that CSparse is very good with matrix compression, since it's 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 [55], 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 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.

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 [64, 48, 45].Figure 6.1 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.

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 6.2 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 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.

6.1 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 the covariance amounts to inverting the information matrix, $\boldsymbol{\Sigma} = \Lambda^{-1}$, where the resulting block matrix $\boldsymbol{\Sigma}$ is no longer sparse. In here, each block $\boldsymbol{\Sigma}_{i,j}$ 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 $\boldsymbol{\Sigma}$. Fortunately, most of the applications require only a few block elements of the covariance matrix, eliminating the need of recovering the full $\boldsymbol{\Sigma}$. 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

Table 5.1: 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**

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 5.1 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 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 maximal. 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.

4 INCREMENTAL SOLVING IN SLAM ++

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 [19, 14]. 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 [50], although it uses Q-less QR factorization rather than the Cholesky decomposition. It was later reimplemented in an experimental branch of g2o[1] 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 closures* 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

[1] Can be found at `https://github.com/RainerKuemmerle/g2o`.

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 [50] 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 [52], 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.

4.1 ALGEBRAIC INCREMENTAL UPDATES OF THE CHOLESKY FACTOR

In this section, the update of the Cholesky factor $R \triangleq \mathrm{chol}(R^\top R)$ 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 (3.3) 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 R. The updated $\hat{R}$ factor and the corresponding r.h.s. $\hat{d}$ can be written as:

$$\hat{R} = \begin{pmatrix} R_{11} & R_{12} \\ 0 & \hat{R}_{22} \end{pmatrix}, \hat{d} = \begin{pmatrix} d_1 \\ \hat{d}_2 \end{pmatrix}. \tag{4.1}$$

From $\hat{\Lambda} = \hat{R}^\top \hat{R}$ and (3.3), the equation (4.1) becomes:

$$\hat{\Lambda}_{22} = \Lambda_{22} + \Omega = R_{12}^\top R_{12} + \hat{R}_{22}^\top \hat{R}_{22}, \tag{4.2}$$

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:

$$\begin{aligned} \hat{R}_{22}^\top \hat{R}_{22} &= \Lambda_{22} + \Omega - R_{12}^\top R_{12}, && (4.3) \\ \hat{R}_{22} &= \mathrm{chol}(\hat{\Lambda}_{22} - R_{12}^\top R_{12}) && (4.4) \\ &= \mathrm{chol}(R_{22}^\top R_{22} + \Omega). && (4.5) \end{aligned}$$

Further in this chapter, (4.4) is referred to as *lambda*-update because it uses parts of the $\hat{\Lambda}$ to update R and similarly (4.5) is referred to as *omega*-update since it directly uses Ω to update R.

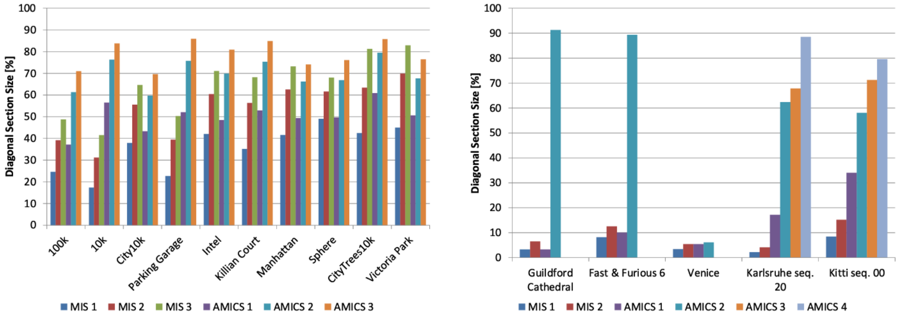


Figure 5.5: Evaluation of nested Schur orderings for up to 3-level nested Schur complements on standard SLAM and up to 4-level on the Schur complements of the BA datasets (the bigger the percentage, the better – it means smaller, denser reduced camera system). The number after the ordering acronym indicates the nesting level.

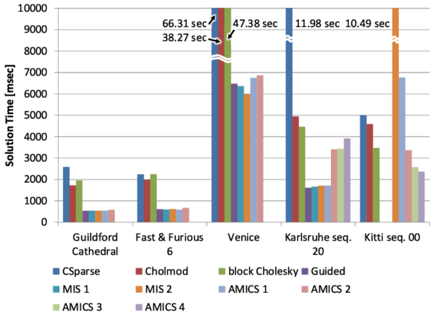
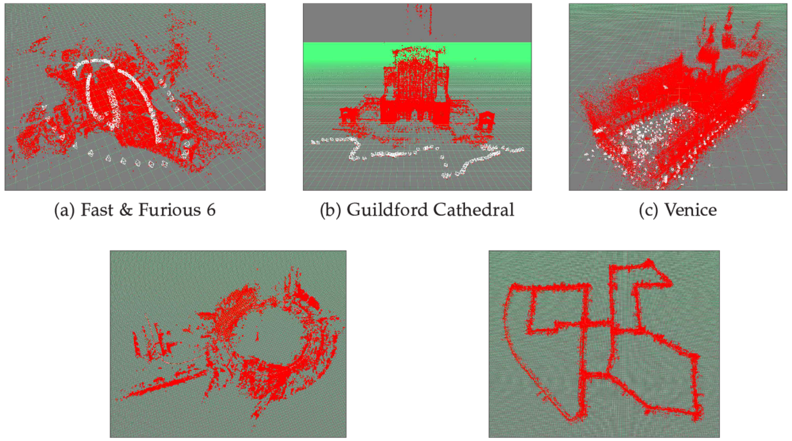


Figure 5.6: Comparison of the effects of ordering on the linear solving on the standard BA datasets. All Schur complement times were obtained using the K20m GPU. The missing times for the *Kitti sequence* 00 dataset were caused by insufficient amount of available memory.

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 5.6. 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 tool[9]. First, the variables were reordered so

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



(a) Fast & Furious 6 (b) Guildford Cathedral (c) Venice

(d) Karslruhe sequence 20 (e) Kitti sequence 00

Figure 5.4: 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.

at 2.30 GHz and sharing 96 GB of RAM, equipped with a single NVIDIA Tesla K20m GPU. The turbo boost function of the Xeon CPUs was disabled for the benchmarks.

The GPUs were employed for dense solving using CULA[8] and for block diagonal inverse, using Cholesky and LU decompositions, respectively. A serial CPU implementation of the sparse block matrix multiplications was employed in (5.2) or (5.3) as it proved to be faster than the off-the-shelf GPU routines. This further demonstrates the efficiency of the block schemes.

The ordering algorithms for Schur complement were also evaluated. In Figure 5.5, 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 left 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 right 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

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

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

$$\hat{R}_{22}^\top \hat{\mathbf{d}}_2 = \boldsymbol{\eta}_2 + \boldsymbol{\omega} - R_{12}^\top \mathbf{d}_1 \,, \tag{4.6}$$

$$\hat{\mathbf{d}}_2 = \hat{R}_{22}^\top \setminus (\hat{\boldsymbol{\eta}}_2 - R_{12}^\top \mathbf{d}_1) \,, \tag{4.7}$$

where \ is linear solving operator; with $\hat{R}_{22}^\top$ 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 $\hat{R}\boldsymbol{\delta} = \hat{\mathbf{d}}$.

4.2 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.

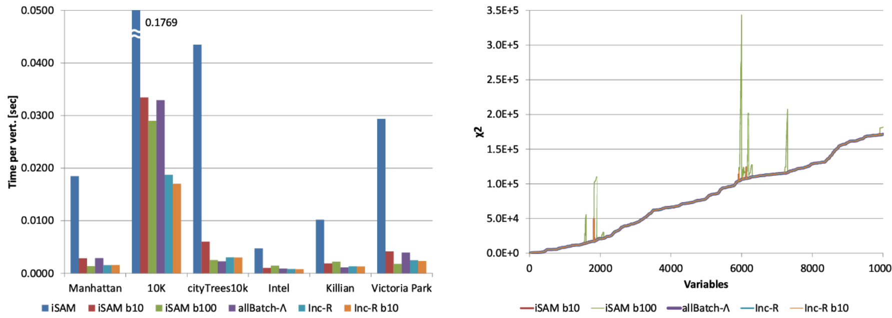
4.2.1 *Efficient Ordering Strategies*

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 [3] 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 [19].

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. Our experiments 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. Applying ordering by blocks instead of elementwise has very small influence in the fill-in of the R factor.

4.2.2 *Fast Update Factorization*

In the increment formula (4.5), 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



(a) (b)

Figure 4.1: a) Time comparison of multiple NLS optimizers, b) Comparison of the χ^2 errors, on the *10k* dataset.

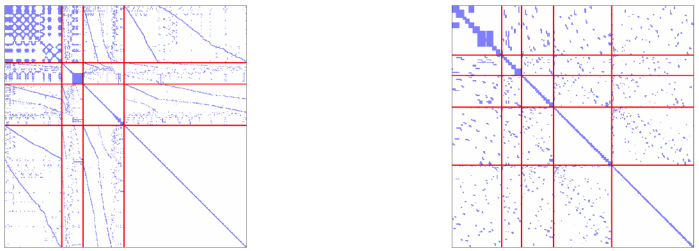
system. On the other hand, here we are interested in the factorization itself. In the proposed implementation, the Cholesky factorization is calculated using CSparse [19] or Cholmod [20] and then *converted back* to a sparse block matrix. This factorization is performed practically at every step and its speed affects the speed of the incremental solver. Fortunately, (4.5) is usually rather small and dense.

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 R is very sparse. Therefore, dense Cholesky is applied for matrices up to 5×5 blocks which occur relatively frequently in (4.5). This Cholesky is further optimized by anticipating the possible combinations of the sizes of 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.

4.3 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 [50], g2o [55], and SPA [53] (a 2D SLAM variant of sSBA). These implementations are easy to use on standard datasets. iSAM2 [51, 52], 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 [51, 52] cannot be used for comparisons since they were measured on a radically different platform.

The evaluation was performed on three standard simulated datasets, *Manhattan*, [67], *10k* and *CityTrees10k*, [49] and on three real datasets, *Intel*, [44], *Killian Court*, [9] and *Victoria park* [65] dataset. The solution for each dataset is shown in Figures 3.1 and 3.2. 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.



(a) Victoria Park (3 levels)

(b) Kitti sequence 00 (5 levels, the bottom-most U, $U^\top$ and D not shown)

Figure 5.3: Examples of nested Schur complements using the AMICS ordering.

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 [33] comprises over 4500 poses.

5.4 EXPERIMENTAL EVALUATION

The experimental evaluations were performed on several datasets which can be seen in Figure 5.4. *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 Canaria[6]. 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 Effects[7]. *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/impart/`. *Venice* is a standard bundle adjustment dataset [55] created from an internet collection of 871 photos of a courtyard adjacent to the San Marco square in Venice, Italy.

Karlsruhe sequence 20 [32] 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 has 967 of them. A similar dataset, *Kitti sequence* 00 of the newer vision benchmark suite by the same authors [33] 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

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

[7] `http://www.dneg.com/`

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

$$\begin{aligned} \text{Schur}(\hat{A}) &= A + \triangle A - (U + \triangle U)(D^{-1} + \triangle(D^{-1}))(U^\top + \triangle U^\top) \\ &= A + \triangle A - (U + \triangle U)D^{-1}(U^\top + \triangle U^\top) - (U + \triangle U)\triangle(D^{-1})(U^\top + \triangle U^\top) \\ &= A + \triangle A - UD^{-1}U^\top - UD^{-1}\triangle U^\top - \triangle U D^{-1}(U^\top + \triangle U^\top) - \\ &\quad\; (U + \triangle U)\triangle(D^{-1})(U^\top + \triangle U^\top) \\ &= \text{Schur}(A) + \triangle A - UD^{-1}\triangle U^\top - \triangle U D^{-1}\hat{U}^\top - \hat{U}\triangle(D^{-1})\hat{U}^\top \qquad (5.6) \end{aligned}$$

and by taking advantage of symmetry:

$$\begin{aligned} D^{-1} &= D^{-\top}\,, && (5.7) \\ (UD^{-1}\triangle U^\top)^\top &= (\triangle U^\top)^\top D^{-\top} U^\top = \triangle U D^{-1} U^\top\,, && (5.8) \end{aligned}$$

it is possible to further simplify (5.6) to:

$$\begin{aligned} \text{Schur}(\hat{A}) &= \text{Schur}(A) + \triangle A - (\triangle U D^{-1} U^\top)^\top - \triangle U D^{-1}\hat{U}^\top - \hat{U}\triangle(D^{-1})\hat{U}^\top \\ &= \text{Schur}(A) + \triangle A - (\triangle U D^{-1}\hat{U}^\top - \triangle U D^{-1}\triangle U^\top)^\top - \triangle U D^{-1}\hat{U}^\top - \\ &\quad\; \hat{U}\triangle(D^{-1})\hat{U}^\top \\ &= \text{Schur}(A) + \triangle A - (E - F\Delta U^\top)^\top - E - \hat{U}\triangle(D^{-1})\hat{U}^\top\,, \qquad (5.9) \end{aligned}$$

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

Due to the highly nonlinear nature of BA, the nonlinear solvers typically take some form of countermeasure to avoid local minima. By employing the Levenberg-Marquardt algorithm [61], a diagonal damping term λ is introduced, yielding a modified normal equation $(\Lambda + \lambda I)\boldsymbol{\delta} = \boldsymbol{\eta}$. This term does change during the solving, causing full-rank incremental updates. For that reason, the Dogleg algorithm [74, 10] is preferred for incremental solving.

5.3 NESTED SCHUR COMPLEMENT

Another interesting option of Schur complement is the possibility to create nested Schur complements. In (5.3), 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

Figure 4.1a shows 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 g2o 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 4.1a shows timing results only for iSAM and for the proposed implementation.

Figure 4.1b 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 4.1b), nicely follows the allBatch-Λ (in violet in Figure 4.1b). 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.

4.4 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 CSparse 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 [52], 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 [51]. In the remaining part of this chapter, an improved incremental algorithm which takes advantage of the sparse block Cholesky factorization from Section 2.1.4 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*[2] 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 [50], 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.

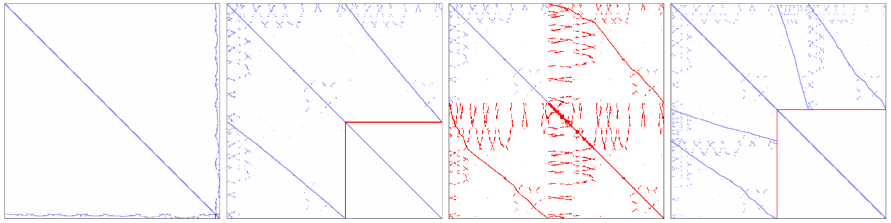
However, this type of QR factorization does not make it possible to reorder the variables: A is ordered using column ordering. Therefore, reordering the columns potentially affects all the rows, making the tracking of changes in the factorization in order to reuse the unaffected parts infeasible. The recently introduced data structure, the Bayes tree [52], 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.

4.5 INCREMENTAL UPDATES OF THE FACTOR USING RESUMED CHOLESKY

Similarly as in Section 4.1, 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 possible to use equations (4.4) and (4.5) 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. The associated cost depends on the size of the update (the number of columns in Ω or 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

[2] 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.



(a) Guided ordering of Λ (b) MIS ordering (c) Clique extended Λ (d) AMICS ordering

Figure 5.2: Finding Schur ordering for landmark SLAM, on the *Victoria park* dataset.

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. Those correspond to the independent cliques in the original graph. The algorithm for finding the maximal (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 5.1. 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 5.2c.

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 5.2d.

5.2 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:

$$\begin{pmatrix} \hat{\Lambda}_{11} & \hat{\Lambda}_{12} \\ \hat{\Lambda}_{12}^{\top} & \hat{\Lambda}_{22} \end{pmatrix} = \begin{pmatrix} A & U \\ U^{\top} & D \end{pmatrix} + \begin{pmatrix} \triangle A & \triangle U \\ \triangle U^{\top} & \triangle D \end{pmatrix}. \tag{5.5}$$

Algorithm 5.1: Finding Maximum Independent Clique Sets.

```
function MICS($w$, $e$)
Require: $w = [w_1, \ldots, w_n]$ is the vector of vertex weights.
Require: $e = \{e_1 \ldots e_m\}$ is a set of edges, where each edge $e_i$ is a pair $(j_i, k_i)$.
    $C = \text{FindCliques}(e)$                          ▷ Use e.g. algorithm of Eppstein et al. [29].
    $P = [\emptyset, \ldots, \emptyset]$                  ▷ $P_v$ is a set of cliques containing vertex $v$.
    for each $c$ in $C$ do
        $w = \left[w;\ \sum_i w_{c_i}\right]$            ▷ Clique weight is a sum of weights of its vertices.
        for each $v$ in $c$ do
            $P_v = P_v \cup c$
        end for
    end for
    for each $c$ in $C$ do
        $V_{adj} = c \cup \{v \mid \exists e_i = (v, u) \in e \wedge u \in c\}$    ▷ Vertices adjacent to clique $c$.
        $C_{adj} = \{P_v \mid v \in V_{adj}\}$           ▷ Cliques adjacent to clique $c$.
        $e = e \cup \{(c, v) \mid v \in V_{adj}\} \cup \{(c, d) \mid d \in C_{adj}\}$    ▷ Add new edges.
    end for
    Return $\text{MaxIndependentSet}(e, w)$              ▷ Use e.g. Tsukiyama et al. [80].
end function
```

brary[5] implements [29] for finding maximal clique sets and [80] 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.

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 [65] dataset (described earlier in Section 3.1, Figure 3.2c), 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 5.2a). 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 5.2b.

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

[5] Can be found at `http://igraph.org/c/`.

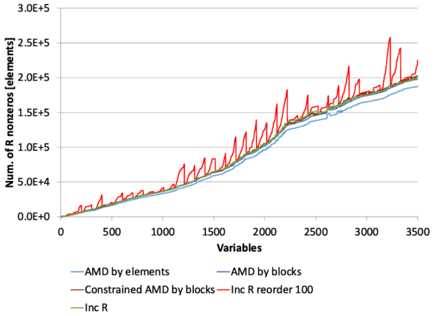


Figure 4.2: Evaluation of several ordering heuristics in terms of nonzero elements compared to the actual number of non-zero elements in the Incremental R algorithm. This is on the *Manhattan* dataset.

update cannot be directly minimized. Ordering the recent variables last as suggested e.g. in [50] 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) [3]. 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}. The following subsection describes how this reordering can be calculated incrementally.

4.5.1 *Incremental Ordering*

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 $O^\top \Lambda O$ are written explicitly, with $R \triangleq \text{chol}(O^\top \Lambda O)$ and $\hat{R} \triangleq \text{chol}(\hat{O}^\top \hat{\Lambda} \hat{O})$.

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 $M \triangleq O^\top \hat{\Lambda} O$ (see Figure 4.3). 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 $M = O^\top \hat{\Lambda} O$ affected by the update, two indices are introduced. The first one, o_{lo} is given by the minimum variable index after the ordering. 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

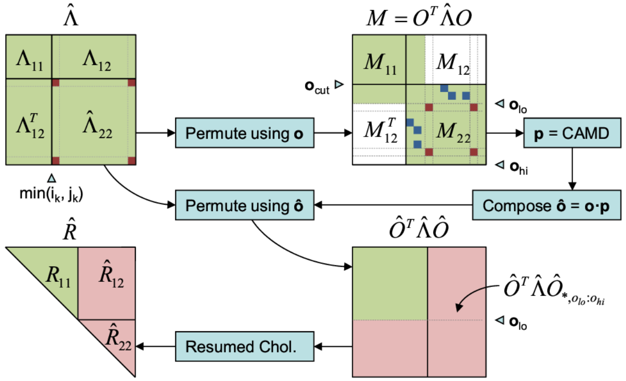


Figure 4.3: Dataflow diagram of incremental block Cholesky factorization. Green parts of the matrices do not change, red parts represent the update and pink represents the parts that will change. White parts are zero. The explanation is simplified to updates involving only two variables. Note that the green parts in $M = O^\top\hat{\Lambda}O$ and in $\hat{O}^\top\hat{\Lambda}\hat{O}$ are unchanged with respect to the previous step.

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 $M_{1:o_{lo}-1,o_{lo}:o_{hi}} = M^\top_{o_{lo}:o_{hi},1:o_{lo}-1}$, which are also affected by this ordering (depicted by the blue blocks in Figure 4.3). 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 $M_{22} \triangleq M_{o_{cut}:o_{hi},o_{cut}:o_{hi}}$ (see Figure 4.3) that satisfies the conditions of being square and not having any nonzero elements above or left from it (so that $M_{1:o_{cut}-1,o_{cut}:o_{hi}} = M^\top_{o_{cut}:o_{hi},1:o_{cut}-1}$ which correspond to the right and bottom portions of M_{12} and $M^\top_{12}$, respectively, are null). The ordering calculated on this submatrix is then combined with the original ordering, 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 $O(o_{hi} - o_{cut})$. 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}. In Figure 4.3 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 (4.5) and

ings used for sparse Cholesky in BA implementations include Multiple Minimum Degree (MMD) [58], AMD [4] or even Reverse Cuthill-McKee (RCM) [16] (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 $LDL^\top$, 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.

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, respectively[3]. 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 *Cathedral* and *Fast & Furious 6*[4] 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 3.5× in all three datasets, compared to the direct solution of normal equations via sparse block factorization.

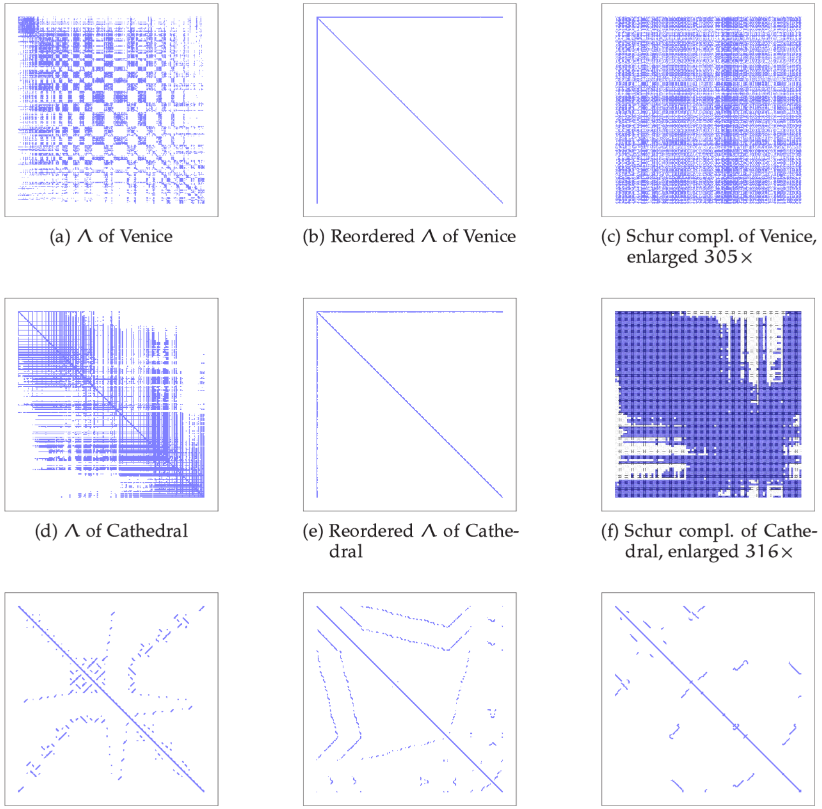
This is because the operations used in Schur complement are simpler ones (for the most part only multiplications and additions) compared to the Cholesky factorization (which requires also a fair amount of divisions and square roots). However, the differences of the cost of these operations is diminished 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.

5.1 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 $U^\top$. 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 [15] li-

[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.



(a) Λ of Venice

(b) Reordered Λ of Venice

(c) Schur compl. of Venice, enlarged 305×

(d) Λ of Cathedral

(e) Reordered Λ of Cathedral

(f) Schur compl. of Cathedral, enlarged 316×

(g) Λ of Parking Garage

(h) Λ of Intel

(i) Λ of Killian court

Figure 5.1: 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.

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 (5.3) 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 $LDL^\top$ decompositions[2]. Densities of as high as 40% occur on e.g. the *Venice* dataset [55] (see Figure 5.1c). Sparse Cholesky solvers have shown about an order of magnitude speedups, especially on large systems and while using a good ordering. The fill-reducing order-

[2] In here, the D is a generic diagonal matrix, other than that in (5.2).

(4.7). Otherwise, the *resumed Cholesky* algorithm is employed. The column Cholesky 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.

4.6 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 the same standard atasets as in the last chapter. 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 chapters.

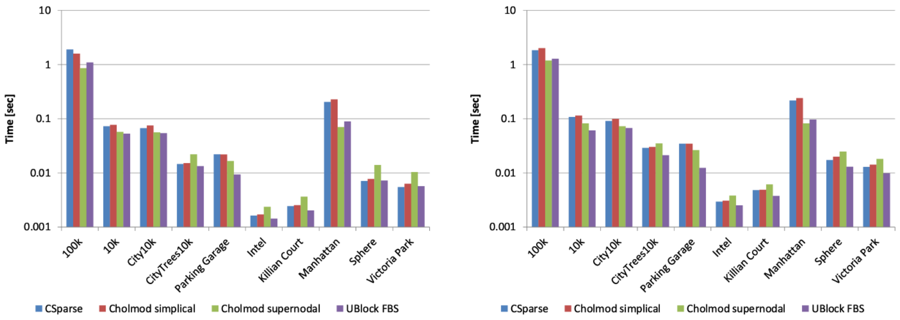
The new library offers the possibility to switch between the "native" block Cholesky (BC) factorization and the Cholesky factorization form CSparse (CS) and Cholmod (CM). Those factorizations are compared on the allBatch-Λ algorithm which is relatively efficient even with elementwise factorization.

4.6.1 *Performance and Accuracy*

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.

The improved implementation reaches the best times for the best accuracy on all evaluated datasets: 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 10th 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 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



(a) (b)

Figure 4.4: Cholesky factorization benchmark on the standard SLAM datasets, a) times of factorization only and b) times of linear solving.

results are shown in Figure 4.4a. 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. Simplical 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 4.4b. Here, backsubstitution is performed along with fill-reducing ordering, Cholesky factorization (and block to sparse matrix conversion for CSparse and Cholmod). This benchmark is relevant because if demonstrates the real performance loss many state of the art NLS solvers pay by not using blockwise representation all the way through.

5 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.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 $\boldsymbol{\theta} = [\theta_1 \dots \theta_n]$ given a set of observations $\boldsymbol{z} = [z_1 \dots z_m]$. 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.

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 $\Lambda\boldsymbol{\delta} = \boldsymbol{\eta}$ can be partitioned as:

$$\begin{aligned} \begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^\top & \Lambda_{22} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{\delta}_1 \\ \boldsymbol{\delta}_2 \end{pmatrix} &= \begin{pmatrix} \boldsymbol{\eta}_1 \\ \boldsymbol{\eta}_2 \end{pmatrix} \quad \text{or} \\ \begin{pmatrix} A & U \\ U^\top & D \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix} &= \begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix}, \end{aligned} \tag{5.1}$$

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 5.1b and 5.1e for examples of matrices from *Venice* [55] and *Guildford Cathedral*[1] datasets: the typical arrow shape shows 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:

$$\mathrm{Schur}(A) \triangleq A - UD^{-1}U^\top . \tag{5.2}$$

This can be used to solve the original system as:

$$\begin{aligned} \left(A - UD^{-1}U^\top\right) \boldsymbol{x} &= \boldsymbol{a} - UD^{-1}\boldsymbol{b}, & (5.3) \\ \boldsymbol{y} &= D^{-1}\left(\boldsymbol{b} - U^\top \boldsymbol{x}\right), & (5.4) \end{aligned}$$

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

[1] can be obtained at `http://cvssp.org/impart/`