MODERN METHODS OF BUNDLE ADJUSTMENT ON THE GPU

R. Hänsch*, I. Drude, O. Hellwich

Dept. of Computer Vision and Remote Sensing, Technische Universität Berlin, Germany (r.haensch, olaf.hellwich)@tu-berlin.de

KEY WORDS: Bundle adjustment, structure from motion, optimization, GPU computing

ABSTRACT:

The task to compute 3D reconstructions from large amounts of data has become an active field of research within the last years. Based on an initial estimate provided by structure from motion, bundle adjustment seeks to find a solution that is optimal for all cameras and 3D points. The corresponding nonlinear optimization problem is usually solved by the Levenberg-Marquardt algorithm combined with conjugate gradient descent. While many adaptations and extensions to the classical bundle adjustment approach have been proposed, only few works consider the acceleration potentials of GPU systems. This paper elaborates the possibilities of time- and space savings when fitting the implementation strategy to the terms and requirements of realizing a bundler on heterogeneous CPU-GPU systems. Instead of focusing on the standard approach of Levenberg-Marquardt optimization alone, nonlinear conjugate gradient descent and alternating resection-intersection are studied as two alternatives. The experiments show that in particular alternating resection-intersection reaches low error rates very fast, but converges to larger error rates than Levenberg-Marquardt. PBA, as one of the current state-of-the-art bundlers, converges slower in 50% of the test cases and needs 1.5-2 times more memory than the Levenberg-Marquardt implementation.

1. INTRODUCTION

In 1958 Duane C. Brown described a method to solve large minimization problems on the basis of least squares (Brown, 1958). This is the first known method for bundle adjustment (BA) and was used to minimize the projection error after estimating point coordinates and camera positions from aerial images. Already shortly after that, it was adapted and applied to close-range photogrammetry. Today it is one of the essential modules in virtually every structure from motion (SfM) pipeline.

SfM aims to recover the position and pose of the cameras as well as the 3D information of sparse scene points from a given set of images. The estimated parameters are prone to inaccuracies caused by wrong correspondences, critical camera configurations (e.g. small baselines), measurement noise, and calibration errors. Furthermore, the optimization is carried out for certain problem subsets (either subset of images (Agarwal et al., 2011) or subset of parameters (Moulon et al., 2013, Wilson and Snavely, 2014)). This leads to solutions that are optimal only for the corresponding subtasks instead of being optimal for the whole task. BA aims to minimize these errors efficiently by performing a global optimization process that considers all cameras and points.

This optimization process implies the formation and solving of equation systems, which becomes particularly computationally expensive for modern datasets involving hundreds of cameras. Therefore, BA is often seen as (one of) the bottlenecks of corresponding reconstruction pipelines.

Over the last years many approaches have been proposed that aim to optimize the efficiency of BA, either on an algorithmic level or by usage of multi-core systems. This paper investigates several methods of BA and discusses theoretical as well as practical approaches to increase accuracy and speed. The latter is achieved by numerical methods and by a GPU based implementation.

The major contributions of this work are threefold: 1) A comparative summary of several state-of-the-art methods for BA, including a discussion on their mathematical foundations, numerical properties, as well as optimizations potentials. 2) An evaluation of their performance with respect to achieved accuracy and speed in one common framework. 3) An open-source implementation of three selected methods for the GPU (available at (Hänsch, 2016)). Each method is optimized with respect to the special requirements of GPU computing under the constraints that the final reconstruction error does not significantly increase and convergence stays guaranteed.

To emphasize the inherent differences of the methods as well as the achieved gain in performance, this work focusses on the use case of 3D reconstructions of very large datasets. The following Section 2 briefly states the problem definition of BA and explains the theoretical basis of three state-of-the-art methods to solve the implied system of equations. Section 3 elaborates both, standard as well as modern optimization approaches of subtasks within BA, and shortly describes available implementations. While Section 2.1 describes mathematical design choices for each of the methods implemented in this work, Section 4 explains details regarding their implementation on GPUs and Section 4.4 discusses corresponding optimization approaches. Section 5 provides the experimental comparison of the different implementations and discusses their individual shortcomings. The last section summarizes the findings of this work and provides an outlook.

2. BUNDLE ADJUSTMENT

The projection of a 3D scene point $X \in \mathbb{R}^4$ to a 2D image point $x \in \mathbb{R}^3$ (both in homogeneous coordinates) is modelled as the perspective projection in Equation 1.

$$x = PX = K[I|0] \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} X \tag{1}$$

The calibration matrix $K \in \mathbb{R}^{3,3}$ describes internal camera parameters such as principle point and focal length, while $R \in \mathbb{R}^{3,3}$ and $t \in \mathbb{R}^3$ model the rotation and translation necessary to transform points from the world coordinate system to the camera coordinate system.

*Corresponding author

In the context of the paper, bundle adjustment means to optimize all m cameras (i.e. $\hat{P}^i, i = 1, ..., m$) as well as all n structure points (i.e. $\hat{X}_j, j = 1, ..., n$) simultaneously starting from an initial estimate given by SfM. This task is modelled as a least squares minimization of a cost function with nonlinearities in the parameterization (the parameter vector θ contains structure and camera estimates, while $\hat{x}(\theta)$ describes the corresponding projection of 3D points in the image of each camera).

$$\min_{\hat{P}^i, \hat{X}_j} \left\{ \underbrace{\sum_{i=1}^{m} \sum_{j=1}^{n} w_{ij} d(x_j^i, \overbrace{\hat{P}^i \hat{X}_j}^{\hat{x}_j^i(\theta)})}_{=:f(\theta)} \right\} \tag{2}$$

Modelling the reprojection error as the squared Euclidean distance d between image point x_j^i and projected point $\hat{x}_j^i$ (in Euclidean coordinates) assumes Gaussian measurement noise. Usually, the set of reprojections is not outlier free (i.e. due to wrong correspondences) which are often prefiltered. To cope with remaining outliers either the measurements are multiplied by a weight w_{ij} which is optimized as well, or more robust cost functions are used (see e.g. (Triggs et al., 2000)).

2.1 NLS Optimization

There are many methods to solve Nonlinear Least Squares (NLS) problems such as Equation 2. This section describes three examples which are often applied for BA. All of them have in common, that they iteratively search for a solution starting from an initial estimate. In the case of BA, this initialization is provided by SfM.

Depending on how fast the sequence of solutions approaches the optimum, the methods are divided into two groups (Triggs et al., 2000). If each iteration (at least potentially) roughly doubles the number of significant digits, the convergence rate is asymptotically quadratic. Such methods are called second order methods. First order methods only achieve linear (asymptotic) convergence. They need more iterations, but usually each iteration is significantly less computationally expensive.

A comparison between first and second order methods can be rarely found within the literature and is completely missing for large problems solved on GPUs. Additionally to the Levenberg-Marquardt algorithm (**LMA**, Section 2.1.2) as second order method, this work implements two first order methods, namely Resection-Intersection (**RI**, Section 2.1.3), and Nonlinear Conjugate Gradients (**NCG**, Section 2.1.1).

2.1.1 Nonlinear Conjugate Gradients Gradient descent methods search the minimum of a given function $f(\theta)$ by starting from an initial value θ_0 and iteratively updating θ_i by Equation 3 using the negative gradient $g_i = \nabla f(\theta_i)$ of f as search direction.

$$\theta_{i+1} = \theta_i - \alpha_i g_i \tag{3}$$

Convergence is guaranteed as long as $\alpha_i > 0$ is not too large and f is convex. There are three principle ways to define α_i for nonlinear functions:

1. Constant learning rate $\alpha_i = \tau$: Convergence is not guaranteed.
2. Exact line search: Based on the current search direction, the optimum is found by solving $\min_{\alpha} f(\theta_i - \alpha g_i)$ (also called *steepest descent*).
3. Inexact line search: The optimal value along the current search direction is approximated by setting α to a high initial value and decrease it iteratively by $\alpha_{k+1} = \tau \alpha_k$ where $\tau \in (0, 1)$ until $f(\theta_k) < f(\theta_{k-1})$.

Conjugate gradients (CG) are proposed in (Hestenes and Stiefel, 1952) for linear equation systems with positive definite system matrix and generalized in (Fletcher and Reeves, 1964). CG approximate the solution of gradient descent by Equation 4. The term s_k is a direction conjugate to the directions of the previous iterations and is calculated by Equation 5. The weighting factor β_k is defined by Equation 6.

$$\theta_{k+1} = \theta_k + \alpha_k s_k \tag{4}$$

$$s_{k+1} = \beta_k s_k - \nabla_\theta f(\theta_k) \tag{5}$$

$$\beta_k = \frac{\Delta\theta_k^T \Delta\theta_k}{\Delta\theta_{k-1}^T \Delta\theta_{k-1}} \tag{6}$$

The difference to gradient descent is that the line search is not applied along the negative gradient, but along the vector s_k which is conjugate to the vector s_{k-1} of the previous iteration (where $s_0 = \nabla_\theta f(\theta_0)$). The vector s_k is conjugate to all previous search directions and points towards the minimum within the subspace defined by $span\{s_0, ..., s_{k-1}\}$.

There are many other approaches to determine β for non-quadratic functions, which promise a better convergence rate (e.g. (Hestenes and Stiefel, 1952, Polak and Ribiere, 1969, Dai and Yuan, 1999)). Each of these possibilities requires an exact line search for a guaranteed convergence. If inexact line search is used, it is not sufficient to ensure $f(\theta_k) < f(\theta_{k-1})$ during backtracking, but stricter conditions have to be fulfilled. The Wolfe condition (Wolfe, 1969) in Equation 7 is commonly used, where $0 < c < 1$.

$$f(\theta_k + \alpha_k s_k) - f(\theta_k) \leq c \cdot \alpha \cdot s_k^T \nabla f(\theta_k) \tag{7}$$

Given an optimal learning rate (and infinite precision), the solution for a quadratic function is found after at most n iterations, where n is the number of parameters of f (i.e. number of dimensions). The practical convergence rate depends mainly on the method (and its accuracy) to calculate α and β. After n iterations, the search directions are not conjugated anymore and the search direction is reset to $-\nabla_\theta f(\theta_k)$.

2.1.2 Levenberg-Marquardt Algorithm The Levenberg-Marquardt algorithm (Levenberg, 1944, Marquardt, 1963) is often used to solve NLS problems, since (for convex functions) convergence is guaranteed and asymptotically quadratic. It is an extension of the classical Gauss-Newton approach. A Newton method solves the problem of nonlinearity by a local quadratic Taylor approximation given in Equation 8, where H is the Hessian matrix.

$$f(\theta + \delta\theta) = f(\theta) + g^T \delta\theta + \frac{1}{2} \delta\theta^T H \delta\theta \tag{8}$$

Instead of performing the complicated search for the minimum of the nonlinear error function, only the global optimum of its local Taylor approximation at an initial estimate θ has to be found. The minimum at $\theta + \delta\theta$ serves as new starting point for the next iteration. The so called Newton step $\delta\theta$ is estimated as $\delta\theta = -H^{-1} g$. In the case of BA the gradient g is given by Equation 9, where $e = x - \hat{x}$, J is the Jacobian, and W the weighting matrix.

$$g = \frac{df}{d\theta} = J^T W e \tag{9}$$

$$H = \frac{d^2 f}{d\theta^2} = J^T W J + \sum_i (e^T W)_i \frac{d^2 f_i(\theta)}{d^2 \theta} \tag{10}$$

If the error e is small or the model close to linear (i.e. $\frac{d^2 f(\theta)}{d^2 \theta} \approx 0$) then $H \approx J^T W J$, which leads to the normal equation (Eq. 11).

$$H\delta\theta = -g \tag{11}$$

An approach exploiting this simplification is called Gauss-Newton method. Without a controlled learning step, convergence is not guaranteed.

The Levenberg-Marquardt algorithm is a standard Gauss-Newton method combined with a controlled learning rate by extending Equation 11 with a damping factor λ, where either $D = I$ (additive extension according to Levenberg) or $D = diag(J^T W J)$ (multiplicative extension according to Marquardt).

$$(H + \lambda D)\delta\theta = -g \tag{12}$$

LMA uses a simple heuristic to control λ: It is increased if an iteration did not reduce the error. Small values of λ result in Gauss-Newton steps, while large values lead to steps in direction of the negative gradient. Since Gauss-Newton has a quadratic convergence close to local minima, λ is kept small in this case.

2.1.3 Resection-Intersection A completely different approach is to divide the problem in multiple, easier subproblems and to alternate between solving each of them. One of the most commonly used methods in the context of BA is *Resection-Intersection* (also called *interleaved Bundle Adjustment*, (Bartoli, 2002, Liu et al., 2008, Lakemond et al., 2013)). The approach is divided into two steps: During the first resection phase, the structure information is kept constant and the problem is solved for all cameras independently for example using the Levenberg-Marquardt algorithm. In the subsequent intersection phase the cameras are kept constant and the problem is solved with respect to the structure.

3. RELATED WORK

3.1 Solving the (augmented) normal equation

BA methods that are based on solving the normal equation (Equation 11), are often superior in performance and dominate practical implementations (e.g. (Triggs et al., 2000)). While a discussion of all possibilities to solve a given equation system is clearly beyond the scope of this paper, this subsection discusses the few methods that are especially suited for BA.

Already D.C. Brown described within the first BA algorithm, that not the complete system is solved, but only a part of it, while the remaining equations are solved by back substitution. The used approach is usually called Schur complement trick.

A square matrix $M \in \mathbb{R}^{q+p,q+p}$ can be decomposed as given in Equation 13, where $A \in \mathbb{R}^{p,p}$ is invertible and $D \in \mathbb{R}^{q,q}$.

$$\begin{aligned} M &= \begin{pmatrix} A & B \\ C & D \end{pmatrix} \\ &= \begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix} \end{aligned} \tag{13}$$

The matrix $S = D - CA^{-1}B$ is called Schur complement of A.

Given an equation system $Mx = b$, where $x = (x_1, x_2)^T$ and $b = (b_1, b_2)^T$, with $x_1, b_1 \in \mathbb{R}^p$, $x_2, b_2 \in \mathbb{R}^q$, x_2 can be computed from $Sx_2 = b_2 - CA^{-1}b_1$. Through back substitution x_1 can be computed from $Ax_1 = b_1 - Bx_2$. For $p \ll q$ these two equation systems are much more efficient to solve than the original system.

If applied to BA (i.e. $M = H$, $x = \delta\theta$, $b = -g$, A is the structure block, D the camera block), the equation system of x_2 is the so called *Reduced Camera System* (RCS).

After the Schur complement, there are two equation systems to be solved. The approach of multiplying with the inverse to solve the equation system is virtually never used in practical applications, due to its lack of numerical robustness. An LU decomposition (equivalent to Gaussian elimination) is more robust, but doesn't exploit the symmetry of the system matrix.

The equation of the back substitution of the Schur complement has a block diagonal form and can be solved blockwise. Each block is positive definite, symmetric, and dense if the matrix was augmented with λ. That is why usually the Cholesky decomposition $M = LL^T$ (L is a lower triangular matrix) is used, which is nearly twice as fast as the LU decomposition (see (Triggs et al., 2000, Agarwal et al., 2010, Konolige and Garage, 2010, Jeong et al., 2012)). The system $Mx = b$ with $M = LL^T$ is solved by forward substitution, i.e. solving $Ly = b$ for y, and backward substitution, i.e. solving $L^T x = y$ for x.

The system matrix of the RCS is symmetric and positive definite, but often not dense, which offers (additionally to the standard way of a Cholesky decomposition) the possibility to use a sparse Cholesky decomposition that exploits the sparsity.

In the case of very large equation systems (hundreds of cameras, thousand points) exact solutions of the normal equation are computationally expensive. In (Triggs et al., 2000) linear CG is recommended to derive an approximate solution. It can be applied to solve the whole normal equation as well as to solve the RCS, as long as the system matrix is symmetric and positive definite.

3.2 Conditioning

The condition number κ of a normal matrix is defined as the ratio of the largest and smallest eigenvalue. BA in general has a bad conditioned cost function (i.e. Hessian matrix). A bad condition leads to elongated ridges around minima, which is especially problematic for first order approaches. The gradient descent methods for example would approach the minimum in a strong zig-zag pattern, which leads to a slow convergence. A solution is to combine the search directions of the current and last iteration (Barzilai and Borwein, 1988). Even for CG methods, a badly conditioned problem leads to slower convergence since the computation of the new search direction requires a high accuracy which suffers the more, the worse the conditioning is. The convergence rate of direct methods (such as Gauss-Newton) does not depend on the conditioning. However, a bad condition can lead to numerical instabilities due to rounding errors.

Preconditioning is often used to ease these problems. An equation system $Ax = b$ with a badly conditioned matrix A is transformed into a system $M^{-1}Ax = M^{-1}b$ with equal minimum, but $\kappa(M^{-1}A) \ll \kappa(A)$.

The choice of a preconditioner that is fast to compute and reduces the condition number sufficiently is a very active field of research, especially for specialized problems such as BA.

Non-block-based CG methods often use the Jacobian conditioner (Equation 14) as well as the *Symmetric Successive Over Relaxation* conditioner (Equation 15) (Saad, 2003), where $H = D + L + L^T$ (D is a diagonal matrix and L an lower triangular matrix) and $0 \leq \omega < 2$.

$$M_{SJac} = diag(H) \tag{14}$$

$$M_{SSOR} = \left(\frac{D}{\omega} + L\right)^{-T} \frac{\omega}{2-\omega} D \left(\frac{D}{\omega} + L\right)^{-1} \tag{15}$$

The entries of the Hessian matrix of BA (Equation 16) are rather blocks than scalar values, which is why block-based conditioners are usually used.

$$H = \begin{pmatrix} U & W \\ W^T & V \end{pmatrix} \tag{16}$$

One common choice is the *Block-Jacobian* conditioner (Wu et al., 2011, Byröd and Åström, 2010, Agarwal et al., 2010), which is easy to invert since the blocks U and V are block diagonal (Triggs et al., 2000).

$$M_{BJac} = \begin{pmatrix} U & 0 \\ 0 & V \end{pmatrix} \tag{17}$$

The work of (Agarwal et al., 2010) uses $M_{BJac} = blockdiag(S)$ or $M_{BJac} = U$ for the RCS, where S is the Schur complement of U. In (Jeong et al., 2012) it is shown, that blockwise conditioners are superior to their scalar alternatives.

In (Agarwal et al., 2010) a conditioner is proposed that is based on the submatrices J_C and J_P of the Jacobian describing camera and point constraints, respectively. The QR decomposition ($J_{C/P} = Q(J_{C/P})R(J_{C/P})$) of these matrices leads to two upper triangular matrices that are used as block diagonal elements:

$$M_{QR} = \begin{pmatrix} R(J_C) & 0 \\ 0 & R(J_P) \end{pmatrix} \tag{18}$$

The inverse is easy to compute since the blocks of the diagonal are block diagonal (Byröd and Åström, 2010).

3.3 Other optimizations

Many BA methods use floating points with double precision to reduce the risk of rounding errors. Implementations on the GPU are often based on single precision (Wu et al., 2011), because double precision needs twice the amount of memory and is slower (Itu et al., 2011). To avoid rounding errors for bad conditioned systems, (Liu et al., 2012) proposes to use double precision for numerically critical operations like matrix inversion or Cholesky decomposition.

The benefits of blockwise processing during conditioning apply to general operations as well. Based on the block structure of the Jacobian matrix, (Wu et al., 2011, Jeong et al., 2012) use the block-compressed row format, which not only saves memory but also increases the speed if matrix vector operations are executed blockwise.

The Hessian and Schur complement matrices are often not needed explicitly. The linear CG method for example only needs the vector $J^T(Jp)$ instead of the whole matrix J^TJ. Using this relation leads to the *Conjugate Gradient Least Squares* method described for example in (Agarwal et al., 2010), which additionally shows that a similar method is possible for the Schur complement.

3.4 Available Implementations

There are three commonly used implementations for BA. All three are based on LMA, but exploit the sparseness of the Hessian matrix in different ways.

The C/C++ project *Sparse Bundle Adjustment* (***SBA***) introduced in (Lourakis and Argyros, 2004) solves the RCS under usage of the Library *LAPACK* (Linear Algebra PACKage). The disadvantage is, that LAPACK does not support solving sparse equation systems.

The C++ implementation of *Simple Sparse Bundle Adjustment* (***SSBA***) (available at (Zach, 2011)) uses the Library CHOLMOD, which does respect the sparseness of the RCS. A direct comparison of SBA and SSBA is missing. Theoretically, the latter should be superior to the first, though. The work of (Konolige and Garage, 2010) implements a BA solution very similar to SSBA, which is indeed significantly faster than SBA.

The C++/Cuda program *Parallel Bundle Adjustment* (***PBA***) for multi-core architectures (CPU and GPU) introduced in (Wu et al., 2011) solves the RCS by conjugate gradients with preconditioner. The work of (Wu et al., 2011) compares PBA to several other BA approaches on very large datasets (more than 100,000 measurements) and shows a 2-3 magnitudes faster convergence rate than algorithms similar to SSBA.

4. IMPLEMENTATION

This section covers implementational details of the three methods described in Section 2.

4.1 Nonlinear Conjugate Gradients

NCG does not need a complete linearization but only the gradient g at the beginning of each iteration. This work additionally computes a blockwise Jacobian conditioner. The diagonal blocks of the Hessian matrix are directly calculated from the status vector instead of computing and saving the Jacobian matrix. The conditioning $M^{-1}g = z$ is applied by solving the linear equation system $Mz = g$ with a Cholesky factorization, which is more stable and efficient than inverting M. Since the block diagonals of M are in general not positive definite, the diagonal is extended with a constant damping factor λ.

A line search based on backtracking is conducted along the search directions. Convergence is ensured by the Wolfe constraint. The convergence of this method largely depends on this line search and especially on the parameters of the Wolfe constraints.

The line search ends if a) the Wolfe constraint (Equation 7) is fulfilled (α gets accepted), or b) the search area along the line defined by α is too small (α is only accepted if error is reduced), or c) numerical problems occur (α is rejected).

The whole optimization ends if the maximal number of iterations is reached, the projection error is below a threshold, or no error reduction was achieved with the last ten iterations.

4.2 Levenberg-Marquardt Algorithm

The LMA implementation of this work follows largely the standard algorithm. At the beginning of each LMA iteration a complete linearization is carried out. The Jacobian matrix, the gradient, as well as the conditioning matrix are calculated on the GPU and saved. Two different solutions are implemented: **LMAH** solves the complete extended normal equation (Equation 12) by conjugate gradients. **LMAS** computes the RCS from the Hessian matrix and the gradient which is solved by conjugate gradients.

The Hessian matrix (or the Schur complement matrix) is calculated implicitly as described in (Agarwal et al., 2010). The disadvantage of this approach is that it has to be recalculated at each CG iteration, which has an negative influence on the iteration duration (especially in the case of the Schur complement).

At the beginning of each LMA iteration, the conditioning matrix is computed, inverted, and saved to memory (in case of LMAS

only the part regarding the camera parameters). Within a CG iteration the conditioning is applied. Contrary to nonlinear CG, the conditioning matrix does not change during a LMA iteration.

When the complete update vector is computed, λ is increased if no reduction of the error was achieved. Otherwise, the update vector is applied by $\theta_{t+1} = \theta_t + \delta\theta_t$, a linearization at the new position is computed, and λ is decreased. The adaption of λ follows (Lourakis and Argyros, 2004) and is based on the gain ratio, i.e. a comparison of the predicted and actual error, which prevents a too strong decrease of λ.

The CG iterations are stopped when the maximal number of iterations is reached, if the ratio of the initial and current remaining error is below a threshold, or the current error is ten times larger than the initial error. The outer iterations terminate if the maximal number of iterations is reached, the projection error is below a certain threshold, no error reduction took place within the last ten iterations, or λ reached its maximal value.

4.3 Resection-Intersection

In a first step an initial blockwise linearization is computed by building the extended normal Equation 12 for single cameras and points. The corresponding equations are solved either only for cameras or only for points depending on whether the iteration number is even or odd. A simple Cholesky decomposition with forward and subsequent backward substitution is used to solve the equation systems.

The diagonal elements are damped by a multiplicative extension $J^TJ + \lambda diag(J^TJ)$ to solve systems with partially indefinite matrices. The damping factor λ is reinitialized after each complete outer iteration. During the LMA optimization λ is increased (decreased) if the error reduction was unsuccessful (successful).

The inner loop is left if either the maximum number of total or successful inner iterations is reached, or the error could not be reduced by a sufficiently large margin.

The outer loop ends if the maximal number of iterations is reached, the projection error is below a predefined threshold, the error was not reduced in ten subsequent iterations, or the parameter λ reached its maximal value.

4.4 Optimization strategies

The individual time performance of the vanilla implementation of all three methods is shown at the top of Figure 1, where the GPU usage is shown in grey under the time axes.

The first row shows three iterations of NCG of different length. The green and purple blocks compute the gradients and conditioning matrix, which is applied in the dark violet block. The dark red (calculating the error vector) and brown (summing the error vector) blocks show the line search and how different its computational complexity can be. Since more than 60% of the time is spent on line search (as the lower half of Figure 1 shows), the focus of further optimizations has to be on this part of the computation.

The timeline of LMA (LMAH is used here as example) in the second row shows three iterations. The linearization is carried out in the beginning (brown for cameras, green for points). Inverting the conditioning matrix is shown in blue. The normal equation is solved by CG. The violet blocks, representing the matrix-vector multiplication $J^T(J\theta)$), indicate that many iterations were necessary to achieve a sufficient reduction of the remaining error.

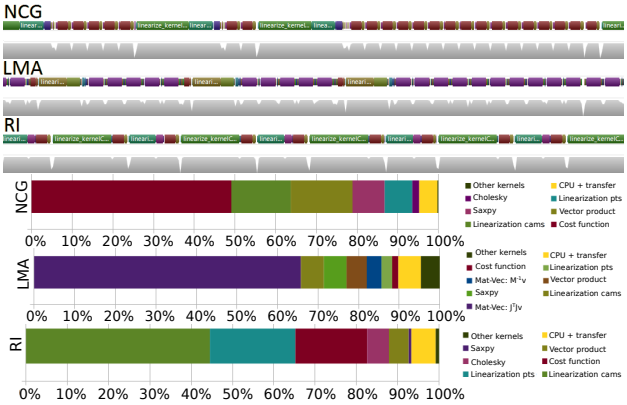

Figure 1. Time of individual processing functions of the three implemented algorithms.

The first two LMA iterations need five CG iterations (i.e. the defined minimum), while the last needs considerably more. The lower part of Figure 1 shows that the computation of the matrix-vector product needs the major part of the processing time despite the rather small number of CG iterations (5-20). The focus is therefore on optimizing the CG approach.

The third row shows nine iterations of RI. Contrary to NCG, the iterations have constant length, where iterations optimizing the cameras take a bit longer since the linearization in this case is computationally more complex. Interestingly, solving an equation takes less time than its creation (shown in violet for points, for cameras not visible due to too short length). This is also shown at the lower half of Figure 1 where 65% of the whole processing time is spent on creating the normal equations and 15% more to compute the error.

There are two principle ways to reduce the runtime of the three implemented methods: A) Reduction of the number of iterations mainly based on numerical approaches and B) Reduction of the iteration duration mainly based on efficient programming on the GPU.

4.4.1 Reduction of iterations In the case of RI, preliminary tests showed that it is beneficial to keep the amount of inner iterations at a minimum. The number of outer iterations is increased due to a less accurate solution, but the total amount of iterations decreases.

In the case of NCG the number of inner and outer iterations can be differentiated as well. The number of outer iterations is decreased by checking the search direction for its potential to reduce the error (as described in Section 2.1.1). Another common approach is the preconditioning of the system for which the block-Jacobian matrix is selected (see Section 3.2).

The reduction of the number of inner iterations basically means to reduce the number of steps during line search. This could be easily achieved by different parameter settings, e.g. by a larger step width. However, this increases the risk to miss the searched minimum. A less strict stopping criterion would end the line search faster, but would uncontrollably increase the amount of outer iterations. This work uses an approach based on the Newton-Raphson method (as described in (Shewchuk, 1994)) which adaptively changes the initial value of α according to Equation 19.

$$\alpha = \frac{g^T\theta}{\theta^TA\theta} \tag{19}$$

Tests showed that most of the line searches were finished after one iteration. There was neither an increase in the number of outer iterations nor a decrease of the accuracy after convergence.

Since LMA can be seen as one of the standard approaches for optimization of nonlinear equation systems, the literature provides many ideas for improvement of which Section 3.1 names a few. The implementation of this work uses an intelligent manipulation of the damping factor λ as described in Section 4.2 to reduce the number of unsuccessful outer (LMA) iterations.

4.4.2 Reduction of iteration duration This section describes how the algorithms presented in Section 2.1 are implemented for execution on the GPU. Since presenting the whole implementation is beyond the scope of this paper, only the most important design decisions are emphasized. The focus lies especially on reducing the time spent during each iteration.

Each nonzero submatrix of the Jacobian matrix representing a camera and/or point block is handled by one thread to exploit the sparse nature of the Jacobian matrix on the one hand and to use blockwise computations as described in Section 3.3.

The follwing list names the most important kernel functions and how they are parallelized:

- $J^T(Jp)$ - Multiplication : per measurement (blockwise)
- J, g, M - Linearization: per measurement (blockwise)
- $f(\theta)$ - Cost function at θ : per measurement (scalar)
- M^{-1}- Inversion: per camera/point (blockwise)
- $Mx = b$ - Solving SLE (Cholesky): per camera/point (blockwise)
- $M^{-1}v$ - Multiplication: per camera/point (blockwise)
- $\alpha\vec{p} + \vec{v}$ - SAXPY-operation: per single value (scalar)
- $p^T v$ - vector product: per single value (scalar)

One reason to cause a suboptimal GPU usage (i.e. less parallel computations as theoretically possible) is that threads use too many registers. Especially the kernels to compute the Jacobian matrix, M^{-1}, and $Mx = b$ need many registers. There are three principle ways to cope with this problem. 1) Start only as many threads as the number of available registers allows. Depending on the given GPU architecture, this can lead to a considerable decrease in efficiency. 2) Ensure an efficiency of 100% by swapping registers to slower memory (register spilling). Depending on the number of swapped registers this is the L1-cache or even the global memory. Especially the usage of the global memory should be avoided, since it is significantly slower and the parallel execution might be decreased by blocking DRAM accesses. 3) Alternatively, multiple threads could handle one block using the shared memory which decreases the amount of used registers.

The first option neither uses the L1 cache nor the shared memory and is the most inefficient one. The third option is complex to be realized and an inefficient access to the shared memory might even have the opposite effect due to access conflicts. This work uses the second option, since all threads are independent of each other. At no point data needs to be shared between threads and the L1 cache (i.e. shared memory) can be completely used for register spilling. There is one single exception where inter-thread communication is necessary: During the computation of the vector product all values need to be summed up in the last step. This is done via parallel reduction (Harris, 2007).

All threads respect the common memory hierarchy of GPUs. Registers are prefered over other memory types and tried to be used as much as possible without decreasing the maximal number of parallel executed threads too much. Nevertheless, especially larger kernels are designed to use as many registers as possible to avoid slow memory access. Despite an occupancy which is sometimes as low as 60-80% of the theoretical possible occupancy, the computational speed is increased due to higher IPC (see also (Volkov, 2010)). L1 cache and shared memory are prefered over global memory and used even if it is not necessary to share data between threads of one GPU block.

The implementations do not use any complex data structures but are solely based on floats. The data is structured such that data access is coalesced. The blocks of the Jacobian matrix for example are ordered in an array as shown below:

$$\begin{aligned} valC &= [CamBlock\#1\,[0]\,, ..., CamBlock\#N\,[0]\,, \\ &\quad\; CamBlock\#1\,[1]\,, ..., CamBlock\#N\,[1]\,, ...] \\ valP &= [PointBlock\#1\,[0]\,, ..., PointBlock\#M\,[0]\,, \\ &\quad\; PointBlock\#1\,[1]\,, ..., PointBlock\#M\,[1]\,, ...] \end{aligned}$$

Since each block is handled within one thread, all parallel access attempts to the first value of the block are coalescing.

To prevent thread divergence no branches based on thread IDs are performed.

Despite the existence of sophisticated CUDA libraries for matrix-vector operations, they are not used in this work. One example is cuBLAS, a library for vector and matrix operations. Its functions are optimized for large and dense matrices, while most tasks in BA involve sparse matrices. The application to dense subproblems would be possible, but the parallelization is optimized for large systems where threads are applied for whole rows/columns instead of small submatrices. Another example is cuSPARSE, a library for multiplication of sparse matrices which even supports a blockwise compressed row format. However, the blocks have to be quadratic, which is not the case for the Jacobian matrix. Furthermore, the library does not provide an option for implicit matrix inversion, which requires explicit computation of matrix inverses.

5. EXPERIMENTS AND RESULTS

5.1 Data and Test System

The following experiments are based on six different, publicly available datasets (Agarwal et al., 2010, Wu et al., 2011). The five datasets of *Trafalgar*, *Dubrovnik-S*, *Venice-S*, *Rome-F*, and *Dubrovnik-F* are based on touristic shots from a public image database. The reconstruction was carried out by an incremental SfM pipeline (Agarwal et al., 2011), which automatically defines a skeletal graph representing images that are sufficient to describe the major part of the 3D scene. A first reconstruction is based on this subset of the images and denoted with the suffix *S*, while the remaining images are attached subsequently and marked by the suffix *F*. Intermediate results are not further optimized.

An additional dataset is provided in (Agarwal et al., 2010), which is based on images taken with a *ladybug* camera mounted on a moving car. Due to the camera movement, images close to each other within the sequence have a high overlap, while images further away do not share any content. The camera-point and point-camera blocks of the Hessian matrix are rather sparse and form a matrix close to a tridiagonal form.

Table 1 summarizes the used datasets with respect to number of cameras, 2D and 3D points, as well as initial mean squared error of the backprojection. Properties regarding the connectivity between the cameras are discussed in (Agarwal et al., 2010).

All tests are run on a Geforce GTX 670 (Kepler architecture) GPU with a compute capability 3.0 and two GByte of (Off-Chip) GDDR5 memory.

Name	#Cam.	#3D Points	#2D Points	MSE
Trafalgar-S	257	65,132	225,911	217.45
Dubrovnik-S	356	226,730	1,255,268	168.26
Ladybug	1723	156,502	678,718	272.79
Venice-S	1778	993,923	5,001,946	102.52
Rome-F	1936	649,673	5,213,733	70.36
Dubrovnik-F	4585	1,324,582	9,125,125	132.16

Table 1. Used datasets

5.2 RI vs. NCG vs. LMA

The three implementations of Resection-Intersection (RI), Nonlinear Conjugate Gradient (NCG) and Levenberg-Marquardt (LMA) are applied to all datasets, the latter in its two variations denoted as LMAH (solving the whole equation system based on the implicit calculation of the Hessian matrix) and LMAS (solving the RCS based on the implicit calculation of the Schur complement matrix).

The optimization is carried out once only for camera pose and 3D points (denoted by "m") and once additionally for focal length and two parameters for radial distortion (denoted by "fr"). Figure 2 shows the progress of the projection errors over time. The left side shows the results for the metric, the right side for the focal-radial case. The difference between the two optimization problems is quite obvious. The corresponding reference files provide the same distortion parameters for all images despite the fact that the images were acquired with many different cameras. Thus an optimization of focal length and radial distortion is beneficial. On the other hand, does the optimization problem obtain more degrees of freedom which allow a closer fit to the data. This does not necessarily mean a better (in the sense of being closer to reality) reconstruction. An exception from this general trend is the ladybug dataset, which was acquired by a single moving camera. Modelling this data as many individual cameras with unshared parameters does not significantly decrease the projection error.

The two LMA variants are clearly superior to the first order methods RI and NCG. LMAS is reaching the smallest error after convergence due to the better conditioning of the Schur-complement matrix (Wu et al., 2011). LMAH generally converges faster thanks to a shorter duration of a single iteration. This is surprising in the sense, that the Schur complement is supposed to speed up the process of solving the equation system. The reason is that the implicit calculation of the Schur complement matrix has to be done in each CG iteration instead of calculating it once, keeping it in memory, and reusing it in each iteration. But this would consume considerably more memory, since the resulting Schur complement matrix is significantly less sparse than the corresponding Hessian matrix. The first order methods descend faster than LMA in the initial iterations. Especially RI is successful in this sense, while NCG has problems to find good initial values for the line search, which has a strong influence on convergence speed and value.

PBA (see Section 3) was tested on the same datasets with two variations to solve the normal equation. The implementations of LMA are similar to PBA. For example is the normal equation

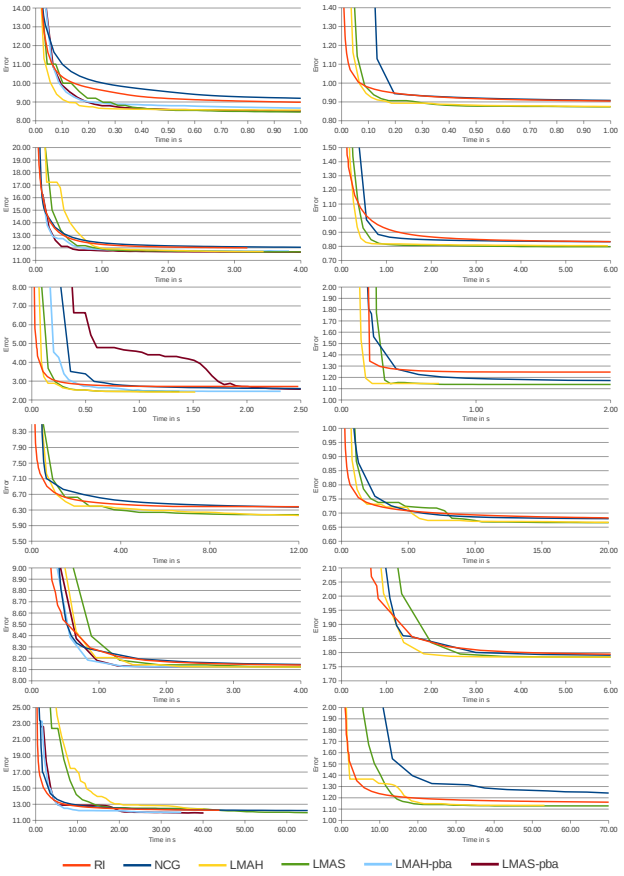

Figure 2. Projection error over time for six datasets (from top to bottom: Trafalgar, Dubrovnik-S, Ladybug, Venice-S, Rome-F, Dubrovnik-F) optimized for camera pose and structure (left) and additionally focal length and radial distortion (right)

solved by CG (with block-Jacobian conditioning) and the Hessian or Schur complement matrix is calculated implicitly in both cases. Differences are the computation of $dot(p, J^T Jp)$, which PBA calculates as $dot(Jp, Jp)$ (Byröd and Åström, 2010) and a different parametrization of the Jacobian with respect to rotations. Since PBA does not allow an optimization of the same radial distortion parameters, it is only applied in the "metric" case (left column of Figure 2). No configuration of PBA was able to optimize the Venice-S dataset. It terminated after six iterations without any error reduction. LMAH converges faster and to a smaller error than PBA for the Trafalgar-S and Ladybug datasets, for the latter by a significant margin. PBA converges faster than the LMA implementations but to the same error for Rome-F. For the Dubrovnik-F dataset PBA is faster and converges to a slightly lower error.

Data	RI	NCG	LMAH	LMAS	PBA
traf	14/14	17/17	35/40	34/39	53
dubs	47/47	55/55	150/177	148/175	276
lady	29/29	35/35	88/104	86/101	156
ven	185/185	219/219	652/712	641/701	1021
rome	167/167	190/190	571/690	563/682	1096
dubf	303/303	349/349	1016/1241	1007/1216	1935

Table 2. Memory consumption of the four implemented methods and PBA in MByte (m/fr)

Table 2 shows the memory consumption of the four implemented methods as well as PBA. The two first order methods do not save the Jacobian matrix and need much less memory. LMAH

needs larger parameter vectors since the whole equation system is solved. LMAS does need more space to temporarily save the Schur complement matrix, but this overhead is neglectable compared to the number of cameras. NCG does need more memory than RI due to the used precondition matrix. PBA needs 1.5-2 times as much memory as LMAH, since it saves data in a 128-bit format. This allows to load four floats at once, but causes the allocation of unneeded memory as well. Furthermore, does PBA save the rotation matrix for each camera instead of the reduced rotation vector and the large vector Jp.

6. CONCLUSION AND FUTURE WORK

Within this work three methods to solve the optimization task of bundle adjustment are discussed, namely the commonly used second order method based on Levenberg-Marquardt in two variations, as well as the two first order methods nonlinear conjugate gradients and alternating resection-intersection. Besides a brief description of their mathematical foundations, special emphasis was put on their numerical properties as well as optimizations potentials.

An open-source implementation of the three selected algorithms for GPUs (available at (Hänsch, 2016)) is used to perform an evaluation of achieved accuracy and speed. In particular the comparison between first and second order methods is missing in the literature, especially in the case of BA from large datasets. The provided implementations show similar behaviour in terms of speed as state-of-the-art bundler PBA, but allow larger freedom with respect to the parametrization and need significantly less memory.

The experiments showed that first order methods reach low error rates very fast, but converge to larger error rates than second order methods.

REFERENCES

Agarwal, S., Furukawa, Y., Snavely, N., Simon, I., Curless, B., Seitz, S. M. and Szeliski, R., 2011. Building rome in a day. *Communications of the ACM* 54(10), pp. 105–112.

Agarwal, S., Snavely, N., Seitz, S. M. and Szeliski, R., 2010. Bundle adjustment in the large. In: *ECCV'10 Part II*, Springer, pp. 29–42.

Bartoli, A., 2002. A unified framework for quasi-linear bundle adjustment. In: *ICPR'02*, Vol. 2, IEEE, pp. 560–563.

Barzilai, J. and Borwein, J., 1988. Two-point step size gradient methods. *IMA Journal of Numerical Analysis* 8(1), pp. 141–148.

Brown, D. C., 1958. A solution to the general problem of multiple station analytical stereo triangulation. Technical Report 43, RCA Data reduction.

Byröd, M. and Åström, K., 2010. Conjugate gradient bundle adjustment. In: *ECCV'10, Part II*, Springer, pp. 114–127.

Dai, Y.-H. and Yuan, Y., 1999. A nonlinear conjugate gradient method with a strong global convergence property. *SIAM Journal on Optimization* 10(1), pp. 177–182.

Fletcher, R. and Reeves, C. M., 1964. Function minimization by conjugate gradients. *The Computer Journal* 7(2), pp. 149–154.

Hänsch, R., 2016. Project website. `http://rhaensch.de/gpu-ba.html`.

Harris, M., 2007. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*.

Hestenes, M. R. and Stiefel, E., 1952. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards* 49, pp. 409–436.

Itu, L., Suciu, C., Moldoveanu, F. and Postelnicu, A., 2011. Comparison of single and double floating point precision performance for tesla architecture gpus. *Bulletin of the Transilvania University of Brasov, Series I: Engineering Sciences* 4(2), pp. 131–138.

Jeong, Y., Nister, D., Steedly, D., Szeliski, R. and Kweon, I.-S., 2012. Pushing the envelope of modern methods for bundle adjustment. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34(8), pp. 1605–1617.

Konolige, K. and Garage, W., 2010. Sparse sparse bundle adjustment. In: *BMVC'10*, BMVA Press, pp. 102.1–102.11.

Lakemond, R., Fookes, C. and Sridharan, S., 2013. Resection-intersection bundle adjustment revisited. *ISRN Machine Vision* 2013, pp. 1–8.

Levenberg, K., 1944. A method for the solution of certain problems in least squares. *Quarterly of Applied Mathematics* 2, pp. 164–168.

Liu, S., Sun, J. and Dang, J., 2008. A linear resection-intersection bundle adjustment method. *Information Technology Journal* 7(1), pp. 220–223.

Liu, X., Gao, W. and Hu, Z.-Y., 2012. Hybrid parallel bundle adjustment for 3d scene reconstruction with massive points. *Journal of Computer Science and Technology* 27(6), pp. 1269–1280.

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

Marquardt, D. W., 1963. An algorithm for least-squares estimation of nonlinear parameters. *SIAM Journal on Applied Mathematics* 11(2), pp. 431–441.

Moulon, P., Monasse, P. and Marlet, R., 2013. Global fusion of relative motions for robust, accurate and scalable structure from motion. In: *ICCV'13*, IEEE, pp. 3248–3255.

Polak, E. and Ribiere, G., 1969. Note sur la convergence de méthodes de directions conjuguées. *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique* 3(R1), pp. 35–43.

Saad, Y., 2003. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics.

Shewchuk, J. R., 1994. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie-Mellon University. Department of Computer Science.

Triggs, B., McLauchlan, P. F., Hartley, R. I. and Fitzgibbon, A. W., 2000. Bundle adjustment – a modern synthesis. In: *International Workshop on Vision Algorithms*, Springer, pp. 298–372.

Volkov, V., 2010. Better performance at lower occupancy. In: *GPU Technology Conference*.

Wilson, K. and Snavely, N., 2014. Robust global translations with 1dsfm. In: *ECCV'14*, Springer, pp. 61–75.

Wolfe, P., 1969. Convergence conditions for ascent methods. *SIAM Review* 11(2), pp. 226–235.

Wu, C., Agarwal, S., Curless, B. and Seitz, S. M., 2011. Multicore bundle adjustment. In: *CVPR'11*, IEEE Computer Society, pp. 3057–3064.

Zach, C., 2011. Simple sparse bundle adjustment. `https://github.com/royshil/SfM-Toy-Library/tree/master/3rdparty/SSBA-3.0`.