Numerical Linear Algebra Project - GPGPU solutions of the Linear Least Squares Problem for Simultaneous Localization and Mapping

Zhaoyang Lv
4th year PhD
zhaoyang.lv@gatech.edu

Samarth Mishra
1st year MS
smishra@gatech.edu

Stefan Stojanov
1st year PhD
sstojanov@gatech.edu

Abstract

An accurate and fast linear direct solver is crucial to the general nonlinear least square problems widely used in computer vision and robotics. In this project, we choose Simultaneous Localization and Mapping (SLAM) as one application of a nonlinear least-square problem, and focus on studying how the linear direct solvers can be boosted by the GPGPU based solutions. We implement QR and Cholesky least square solvers in both single precision and double precision, and compare their performance w.r.t. the its equivalent direct solvers implemented in Eigen C++ library, which is used as a standard in this application. We provide a full analysis of these implementations in terms of both accuracy and speed, tested in a SLAM dataset [2]. Our experiments indicate the GPGPU based solutions can bring significant speed up in such sparse matrix factorization problems.

1. Introduction

Many problems in computer vision and robotics can be formulated as structured graphical models, such as structure-from-motion (SFM), simultaneous localization and mapping (SLAM), camera calibration, image denoising, etc. One classical representative of such problem is simultaneous localization and mapping (SLAM), which is to optimally estimate the camera poses and 3D positions of all features in the scene.

One common approach to address this problem is 'Bundle Adjustment' which is to solve the camera poses and 3D points based on their constraints as a non-linear least square problem [7]. [4] stated that the Maximum a Posteriori (MAP) Inference of such problems is equivalent to solving a least-square problem in sparse linear algebra. This approach has enabled large scale dense mapping of the world, accurate robot localization among other applications. To scale the application of this method to a large environment or to make it work in an online fashion with low-latency, it is worth exploring a fast and accurate solution.

A standard method solve the nonlinear least-squares problems via an iterative method starting from a suitable initial estimate. At each iteration, a linearized least-square problem is solved. The problem complexity is determined by the iterative methods, e.g. Gauss-Newton (GN), Steepest Descent, and the direct solver to solve the linearized least-square problem, e.g. Cholesky or QR Decomposition. It is also worth noting that the matrix formulation in such problems is highly sparse, which is the case in many other computer vision problems. There are substantial research efforts in accelerating these algorithms by exploiting sparse matrix factorization. However, the complexity of solving the problem sequentially is still highly subject to the number of measurements and unknowns.

In recent years, there has been significant progress in highly parallel general purpose GPU (GPGPU) techniques that accelerate many numerical methods, thus enabling real-time solutions. Although there are extensive existing efforts in improving the speed of least-square solvers on CPUs, very few approaches in the computer vision domain try to address the least square problem by leveraging GPGPU techniques. It is also well-known that the matrix formulations of many vision problems including SLAM have special properties in terms of sparsity, incrementally increasing matrix size, etc. Kaess et al. [6] discovered that an incremental QR factorization based on Givens rotations can effectively provide a solution in constant time for SLAM. Meanwhile, there are several works that study sparse QR factorization leveraging GPGPU, but none of these approaches address these problems in the vision domain.

In this project, we explored the GPGPU techniques that can impact the general sparse matrix factorization, and can benefit the least-square solvers. We will start from the nonlinear least-square problem in the simultaneous localization and mapping (SLAM) and formulate the linearized least-square equations. We will focus on implementing and benchmarking the GPU versions of sparse least-square solvers v.s. the commonly used CPU solvers in C++ Eigen implementation [5]. Our benchmark shows that current GPU implementations can deliver significantly faster sparse matrix solutions without losses in accuracy.

2. Background and Related Work

2.1. Simultaneous Localization and Mapping

SLAM or simultaneous localization and mapping is one of the most important requirements of autonomous robotic agents. More specifically, this requires the robot to (1) autonomously navigate and explore unknown places and (2) allow for the use of the robot's presence in the environment to generate a map of an unknown place for later use. A smoothing approach to SLAM involves not just the most current robot location, but the entire robot trajectory up to the current time. SAM—Smoothing and Mapping [4] is the *full SLAM* approach optimally estimating the entire set of sensor poses along with the parameters of all features in the environment, which also has a long historical connection to photogrammetry where it is known as bundle adjustment and computer vision, where it is referred to as structure from motion.

Dellaert et al. [4] describe how the optimization problem associated with full SLAM can be concisely stated in terms of sparse linear algebra. The full SLAM problem is to estimate the entire trajectory $X = \{x_1, x_2, ..., x_n\}$ and the map $L = \{l_1, l_2, ..., l_m\}$, given the measurements $Z_{z_1,z_2,...,z_k}$. In this paper, instead of using a full SLAM setting, we will only consider the optimization problem that only estimates the entire trajectory X, by marginalizing all the other information into the pair-wise constraint $f(x_{i-1}, x_i)$, which is also termed as the *Pose-Graph problem*. The state X can be solved by obtaining a maximum a posteriori estimate resulting in a nonlinear optimization problem:

$$X = \underset{X}{\operatorname{argmin}} \sum_{i=1}^{N} \|f(x_{i-1}, x_i)\|_2 \tag{1}$$

Solving the nonlinear optimization problem is possible using an iterative method e.g. Gauss Newton Method or Newton's Method. At each iteration, we update $X^{k+1} = X^k + \delta X$, where δX is the solution of the following sparse linear system:

$$\mathbf{J}\delta X = \mathbf{r} \tag{2}$$

Here $\mathbf{r}$ is the residual vector and $\mathbf{J} = \mathbf{J}(X, f)$ is the Jacobian matrix of pairwise constraints $f(x_{i-1}, x_i)$ w.r.t. X linearized at each iteration X^k. The vector δX is the solution of the least-square problem:

$$\delta X = \operatorname{argmin} \|\mathbf{A}\delta X - b\|_2 \tag{3}$$

It is worth to note that matrix A is a normally very sparse matrix (as evident in Figure 3) and over-determined $m \times n, (m > n)$. When $\mathbf{A} = \mathbf{J}, b = \mathbf{r}$, we can apply a QR factorization to the problem and solve it as $\|\mathbf{A}\delta X - b\|_2 = \|R\delta X - d\|_2 + \|e\|_2$. When $\mathbf{A} = \mathbf{J}^\top \mathbf{J}$ and $b = \mathbf{J}\mathbf{r}$, we solve the normal equations with Cholesky factorization.

2.2. Complexity of Sparse Linear Solvers

Since $n < m$ in the general case, the Cholesky factorization enables us to solve the problem much more efficiently with a much smaller scale. Cholesky factorization with normal equations takes $(m + \frac{n}{3})n^2$ flops and Householder QR factorization takes $2(m - \frac{n}{3})n^2$ flops to solve the linear least squares problem. Note that these flop counts are both for dense matrices, where in practice it is evident that the matrices are often sparse. It is important to note the advantage of QR factorization due to its numerical stability.

Another property is that the dimensions of this matrix grow with the number of states, landmarks and measurements used to perform SAM, emphasizing that the use of the sparse characteristics of this in solving the least squares problem is crucial. Nowadays there are strong requirements for low-latency SLAM systems, which would require fast real-time solutions regardless of the problem scale. However, as the problem size goes larger, the complexity of linear least-square problem, which is $(m + \frac{n}{3})n^2$ w.r.t. the number of unknowns in the general Cholesky factorization, is the biggest computation cost.

In this project, we explore methods for linear least squares to evaluate the gains possible in applications such as SLAM/SFM due to using GPUs and exploiting sparsity. Our results show least-squares based approaches can be greatly boosted with the prevalent usage of GPUs.

2.3. Existing GPU Least Squares Solvers

There has been a significant effort towards developing techniques that exploit the inherent parallelism in numerical problems implemented on powerful devices that support parallelism very well such as GPUs. This is evident in the toolkits developed for these devices, with CUDA toolkit provided by NVIDIA that includes the cuSolver library. Specifically relevant to our project is the `cuSolverSP` subset of cuSolver, which is a library specifically developed for solving sparse linear systems. Additionaly, there are Givens rotations included in the cuBLAS library which we used when exploring the Givens rotation based methods for sparse QR decomposition.

3. Our Approach

We establish benchmark performances of different CPU and GPU implementations for the solutions of Linear Least Squares problem on sparse matrices. We use the Eigen C++ library for the CPU implementations and the CUDA cuSolverSP library for the GPU implementations. All these sparse matrix implementations use the CSR/CSC sparse matrix format.

3.1. CSR and CSC sparse matrix formats

The *Compressed Sparse Row* (CSR) format represents a matrix **A** using three one-dimensional arrays, that respectively contain nonzero values, the extent of rows and column indices [8]. The three one-dimensional arrays (M, IM, JM) store a sparse matrix $m \times n$ matrix **A** as (Let NNZ be the number of non-zero elements in **A**) :

- The array M of length NNZ stores the non-zeros elements of **A** in row-major order.
- The array IM is of length $m + 1$. The first m elements of this array store the locations in M of the first non-zero element of each row of **A**. The last element stores the value NNZ (which is where the first non-zero element of the next row would be, had there been a next row).
- The array JM of length NNZ stores the column index in the matrix **A** for each of the elements in M.

As an example, the CSR representation of the matrix

$$\mathbf{A} = \begin{pmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix}$$

can be written as:

```
 M = [ 10 20 30 40 50 60 70 80 ]
IM = [  0  2  4  7  8 ]
JM = [  0  1  1  3  2  3  4  5 ]
```

The *Compressed Sparse Column* (CSC) format is very similar to the CSR format. If (M, IM, JM) = CSC(A), and the elements of $\mathbf{A}$ are stored in column-major order in M, JM stores the row-indices of each element in M, and IM indexes into M, to give the locations of the first non-zero element of each of $\mathbf{A}$'s columns. The sizes of M, IM and JM are NNZ, $n+1$ and NNZ respectively, where $\mathbf{A}$ is a matrix of size $m \times n$ and NNZ is the total number of non-zero elements in $\mathbf{A}$. It is easy to see that CSR($\mathbf{A}$) = (M,IM,JM) = CSC($\mathbf{A}^\top$).

3.2. Method and Benchmark Algorithms

We use an existing implementation of Nonlinear optimization based on a factor graph refering to [4], which provides the linearization of the constraints. By using this implementation, we can derive the Jacobian matrix described in Equation 2. Note that the study of nonlinear optimization is outside the scope of this project. Forthe remaining discussion, we will fix the same linearization step and parameters, and only change the implementation of the linear least-square problem of the linearized matrix in Equation 2.

We implement five least-square solvers in C++ by utilizing C++ Eigen Library and NVIDIA CuSolver Library. For all the CUDA implementations, we tried algorithms using both single precision and double precision. Following are the algorithms we include in our benchmark :

- **CPU** LAPACK Sparse QR Decomposition: Using the `Eigen::SparseQR` class which uses a sparse Householder QR decomposition method.

- **CPU** LAPACK Sparse Cholesky factorization : Using the `Eigen::SimplicialLDLT` class which returns the cholesky factorization of a sparse symmetric positive definite matrix. The solution is used to solve the system of normal equations $\mathbf{J}^\top\mathbf{J} = \mathbf{J}^\top\mathbf{r}$.

- **CPU** Conjugate Gradient Method: `Eigen::ConjugateGradient` class. Iterative linear least-square solvers is often used to solve large scale problems and claimed to have faster convergence. We found that the class conjugate gradient method does not give good convergence, and the method only works with preconditioning using incomplete Cholesky.

- **GPU** cuSolverSP Cholesky factorization: Using the `cusolverSpScsrlsvchol` function (for single precision) and `cusolverSpDcsrlsvchol` function (for double precision) for solving the system of normal equations.

- **GPU** cuSolverSP QR Decomposition: Using low level cuSPARSE functions for double precision `cusolverSpDcsrqrSetup`, `cusolverSpDcsrqrFactor` and `cusolverSpDcsrqrSolve` and the corresponding functions for single precision. cuSPARSE uses a sparse Householder QR decomposition method.

3.3. Parallelism using CUDA kernels

CUDA GPUs can support several parallel threads (~ 2048) on each of multiple different Streaming Multiprocessors. Hence, parallelizing a great amount of lower level operations can lead to a significant speedup on a GPU.

A CUDA kernel is a specialized function that can be executed by one such thread on a GPU, and has access to GPU device memory. CUDA kernels are called by a piece of code running on the CPU through an instruction that specifies the number of parallel threads and blocks of threads to be used.

3.4. Implementing QR Givens : Approach and Problems

Inspired by the sparsity in the matrix representation and Givens rotations in parallel execution, we also implemented a QR Givens method in CUDA. Note that this implementation might not be very efficient on a sparse matrix in the CSR/CSC format because of the constant updates to the number of non-zero elements during computation. This would lead to memory allocations after every Givens rotation is applied—a very time consuming process in comparison to the arithmetic operations. Hence, this method operates on dense matrices as shown in algorithm 1. Note that here, the upper triangular matrix **R** is accumulated in **A** and $\mathbf{Q}^\top b$ overwritten in b. Note that both the matrices **A** and b reside in the GPU memory.

A GPU implementation involving independent row rotations being done in parallel [1] can be written, but is beyond the scope of current work. In our implementation, we try to reduce work by selective rotation when necessary. Note that this requires a zero-check as in line 7 of algorithm 1. There are two approaches that are possible here. Performing the check on the CPU is the first approach, after which a CUDA kernel may be called to modify **A** with the rotation. This zero-check requires an element copy from GPU device memory to host (CPU) memory. However, this copy step bottlenecks the computation, and hence, this approach is not very efficient.

A second approach is to launch parallel CUDA threads and perform the steps inside the loops (lines 5-11 of algorithm 1) in a CUDA kernel. This requires syncing of different threads after the computation of (c, s), because there is a potential race condition, in accessing the elements of **A** and modifying them. This approach did lead to significant speedup as compared to the first one. However, it was not good enough to compete with any of our other GPU benchmarks. Hence, this algorithm was dropped entirely from our list of benchmarks.

Algorithm 1 QR Givens for LS Problem

```
1:  $\mathbf{A}$ : An $m \times n$ matrix
2:  $b$ : An $m \times 1$ vector
3:  for $i = 1$ to $n$ do
4:      for $j = m$ downto $i + 1$ do
5:          $a_1 \leftarrow \mathbf{A}(j, i)$
6:          $a_2 \leftarrow \mathbf{A}(j - 1, i)$
7:          if $a_2 == 0$ then
8:              continue
9:          end if
10:         Compute Givens rotation $(c, s)$ using $(a_1, a_2)$
11:         Apply above rotation on rows $j$ and $j - 1$ of $\mathbf{A}$ and $b$
12:     end for
13: end for
14: Solve the equation $\mathbf{A}(1 : n, 1 : n)x = b(1 : n)$ by back substitution
```

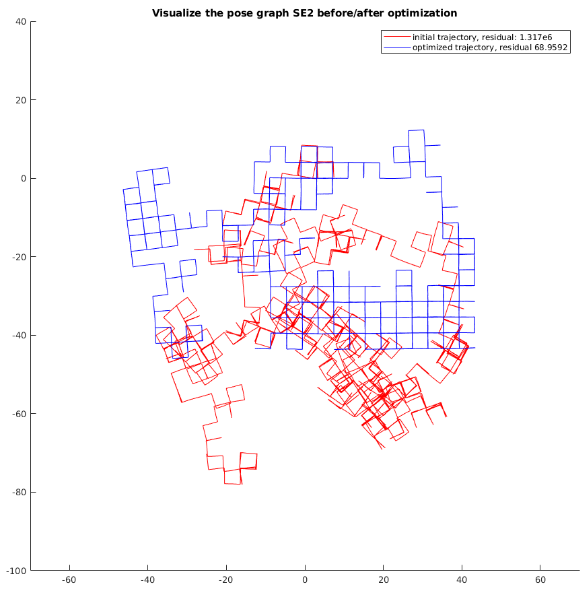

Figure 1. Illustrating the pose graph solved by the Gauss-Newton method at initial state (in red) and the output solution after optimization has been performed (in blue).

4. Experiments

Data and experiment setting We used data from 2D Pose Graph Optimizations [3] to get the LS problems for our study. To exclude the effect of nonlinearity effect in optimization, we specifically used the `m3500` pose graph data from [2], which can be robustly solved via the Newton's method. The method turns to Gauss Newton approach when we are using solving the normal equation with Cholesky factorization for $\mathbf{J}^\top\mathbf{J}\delta X = \mathbf{J}^\top b$. In Figure 1, we show the initial pose trajectory measured from odometry sensors (in red), and the optimized trajectory after optimization (in blue). The optimized trajectory is approximately the same as the ground truth. For all the linear least-square solvers we implemented, they all can converge to the same solution, but differs in execution time and the number of iterations.

Our experiments were performed on an Intel i7-6850K Broadwell CPU for CPU benchmarks and a NVIDIA TITAN X GPU for the GPU benchmarks. The iterative Gauss-Newton optimization method was stopped when the change in error between consecutive iterations was less than $\varepsilon = 10^{-6}$.

Sparsity of problem For the `m3500` pose graph, the linearized system solved at each iteration is 16362×10500 where e.g. at the first iteration $54,533$ elements out of the $171,801,000$ elements of the matrix were nonzero, indicating a highly sparse problem. The sparsity characteristics of the problem are further evident in Figure 3 and Figure 4.

Benchmark For each of the methods we benchmarked speed and accuracy of the linear least square solvers, and for the GPU implementations we evaluate both single precision and double precision versions of the algorithm. The comparison of single precision against double precision on GPUs is of particular interest since fast single precision GPU hardware e.g. NVIDIA GTX series is multiple-fold less expensive than fast double precision GPU hardware e.g. NVIDIA Tesla series. The residual of the linearized system is used to measure the rate of convergence and accuracy.

Table 1. A time comparison among the various GPU and CPU based least squares solvers. SP indicate single precision and DP indices double precision. We did not include Eigen QR timing because it takes too long to generate the solution (2 hours per iteration). It is evident that the GPU based least squares implementations are significantly faster.

Method	Eigen CG	Eigen Cholesky	cuCholesky SP	cuCholesky DP	cuQR SP	cuQR DP
Time	6.6 min	49 s	3481 ms	**2637 ms**	40 s	46.4 s
# Iterations	6	6	22	6	6	6

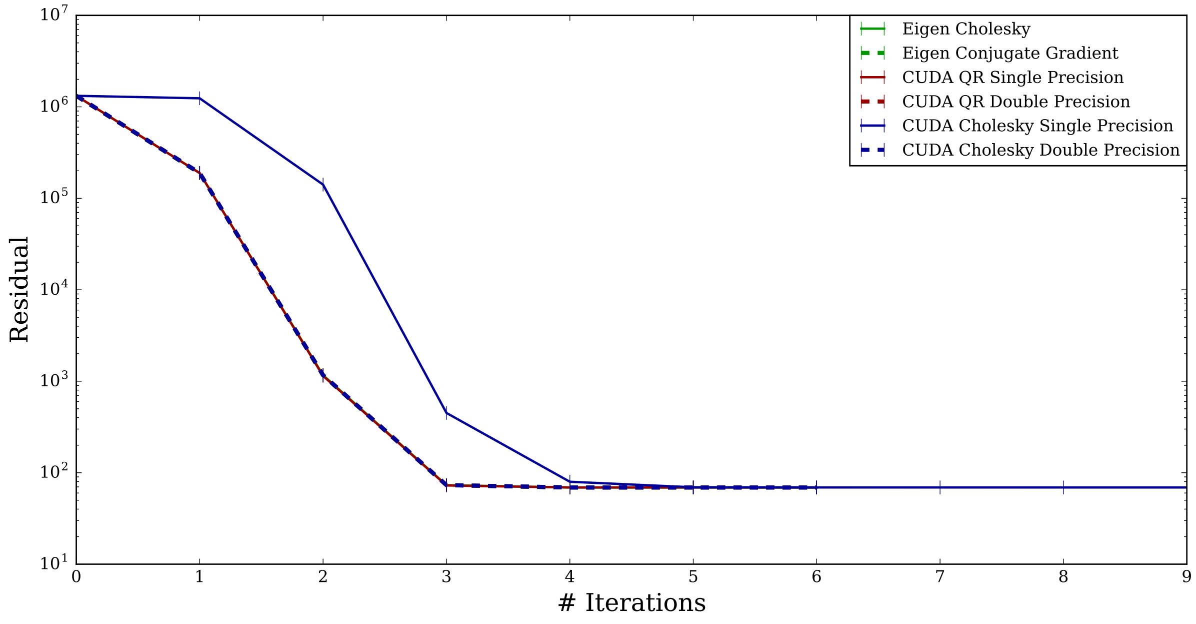

Figure 2. All methods reduce the residual in a similar manner over multiple iterations. Note that the single precision Cholesky solver runs for 22 iterations; this plot has a shorter x axis for the purpose of clarity. We find that the double precision implementation of direct solvers produce the same results as Eigen implementations. Single precision implementation generates the solution slightly faster in each iteration, but provides less accurate solution and takes more iterations for Newton's approach to converge.

5. Discussion

From Table 4 it is evident that using GPU hardware for sparse linear system solving results in better exploitation of the inherent parallelism in such problems. GPU based implementations outperform CPU based implementations by a wide margin. The performance of the GPU Cholesky implementation, that is cuCholesky, implementation provides empirical evidence of the numerical instability of the normal equations approach. It is known that Cholesky factorization is prone to numerical instabillity due to a larger condition number of LHS matrix induced by the normal equations, which is exacerbated by using single precision arithmetic. The single precision cuCholesky took longer (22 iterations) to reach the stopping criterion and had the residual drop slower than other methods. We therefore have shown that using GPU hardware to solve linear least squares problems as a part of the larger simultaneous localization and mapping problem is a very attractive solution. This is especially important since fast computation of these subproblems allows for a larger total number of measurements to be used for the factor graph.

Although for this specific problem, QR factorization and Cholesky based least squares methods do not vary greatly in numerical stability through our experiments we have shown that using GPUs can potentially

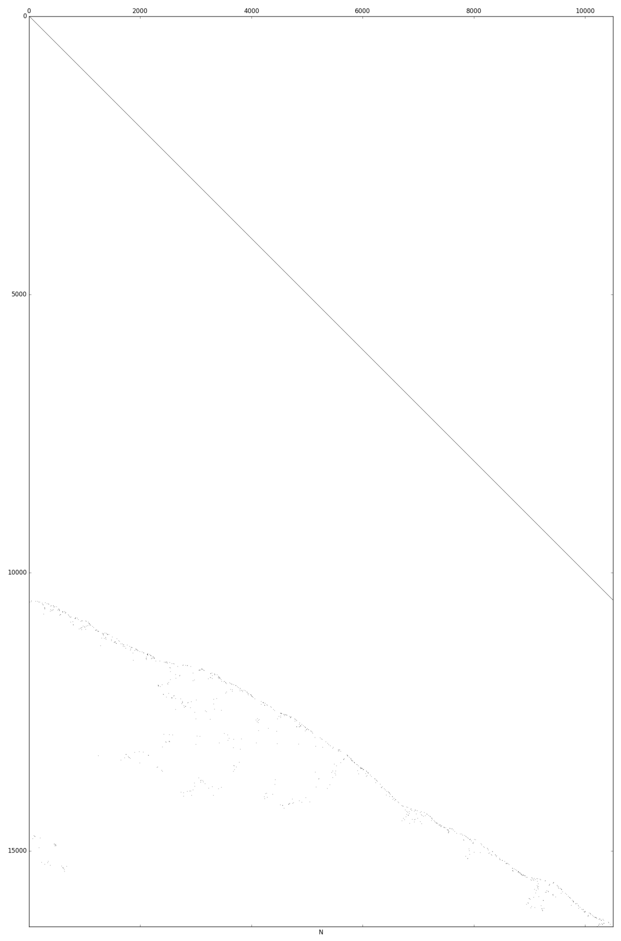

Figure 3. Illustrating the highly sparse structure of the matrix $\mathbf{A}$ in the linear least squares problem. Number of nonzero elements is $54,533$ elements out of $171,801,000$.

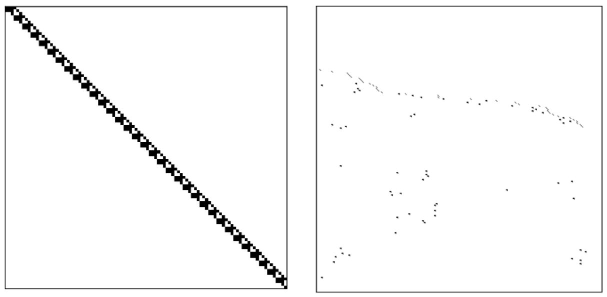

Figure 4. Further evidence of the highly sparse structure of the matrix A in the linear least squares problem. The left image illustrates the structure of $\mathbf{A}[9000:9100, 9000:9100]$ and the right image illustrates$\mathbf{A}[2500:3000, 11500:12000]$

allow for significantly faster and numerically stable QR factorization based solutions for other applications.

6. Conclusion

In this project we implement and apply different linear direct solvers utilizing CPU and GPU to a nonlinear optimization problem and provide an analysis of their comparisons. This analysis mainly brings to light the speed improvement brought about by implementing sparse matrix algorithms on the GPU. The different benchmarks help gauge the relative performance of different algorithms for solving the sparse matrix least squares problem.

References

[1] R. Andrew and N. Dingle. Implementing qr factorization updating algorithms on gpus. *Parallel Computing*, 40(7):161–172, 2014. 5

[2] L. Carlone. `https://lucacarlone.mit.edu/datasets/`. 1, 6

[3] L. Carlone and A. Censi. From angular manifolds to the integer lattice: Guaranteed orientation estimation with application to pose graph optimization. *IEEE Transactions on Robotics*, 30(2):475–492, 2014. 6

[4] F. Dellaert and M. Kaess. Square root sam: Simultaneous localization and mapping via square root information smoothing. *The International Journal of Robotics Research*, 25(12):1181–1203, 2006. 1, 2, 4

[5] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. 2

[6] M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Incremental smoothing and mapping. *IEEE Trans. on Robotics (TRO)*, 24(6):1365–1378, Dec. 2008. 2

[7] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon. Bundle adjustment — a modern synthesis. In B. Triggs, A. Zisserman, and R. Szeliski, editors, *Vision Algorithms: Theory and Practice*, pages 298–372, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. 1

[8] Wikipedia. `https://en.wikipedia.org/wiki/Sparse_matrix`. 3

Appendix A. Code

Following are the selected files in the implementation containing our LS solvers :

- minisam/linear/cuDirectSolvers.h

```
#pragma once

#include <Eigen/Core>
#include <Eigen/SparseCore>

#include <cusolverSp_LOWLEVEL_PREVIEW.h>
#include <cusparse_v2.h>
#include <cusolverSp.h>
#include <cuda_runtime.h>


#ifdef USE_DOUBLE_PRECISION
#define TYPE double
#else
#define TYPE float
#endif

namespace minisam {

class CudaDirectSolver {
    // cholesky ordering
    enum {
      CHOLESKY_NO_ORDERING,
      CHOLESKY_SYMRCM,
      CHOLEKSY_SYMAMD
    };

public:

  CudaDirectSolver();

  CudaDirectSolver(const Eigen::SparseMatrix<double>& A,
   ↪  const::Eigen::VectorXd& b, const bool normal=true);

  ~CudaDirectSolver() {
    free_all();
  }

  void solve_cholesky(Eigen::VectorXd &x);

  void solve_spQR(Eigen::VectorXd& x);

  void compute_residual();

private:

  cusolverSpHandle_t cusolverSpH; // reordering, permutation and 1st LU
   ↪  factorization
  cusparseHandle_t   cusparseH;   // residual evaluation
  cudaStream_t stream;
  cusparseMatDescr_t descrA; // A is a base-0 general matrix

  csrcholInfo_t d_info; // opaque info structure for LU with parital
   ↪ pivoting

  // device sparse matrix in CSR(A)
  int *d_csrRowPtrA; // <int> n+1
  int *d_csrColIndA; // <int> nnzA
  TYPE *d_csrValA; // <double> nnzA
  TYPE *d_x; // <double> n, x = A \ b
  TYPE *d_b; // <double> n, a copy of h_b
  TYPE *d_r; // <double> n, r = b - A*x

  // host
  TYPE *h_x; // <double> n,  x = A \ b

  int rowsA; // number of rows of A
  int colsA; // number of columns of A
  int nnzA; // number of nonzeros of A
  int baseA; // base index in CSR format

  bool is_normal;

  void initialize(const Eigen::SparseMatrix<double> &A,
                  const Eigen::VectorXd &b);

  void free_all();

  Eigen::SparseMatrix<double> csc2csr(const Eigen::SparseMatrix<double>&
   ↪ A) {

    if (is_normal) {
      colsA = A.cols();
      rowsA = A.rows();
      return A;
    }
    else {
      colsA = A.cols();
      rowsA = A.rows();
      return A.transpose();
    }
  }
};

}
```

• minisam/linear/cuDirectSolvers.cpp

```
#include "cuDirectSolver.h"

#include <iostream>
```

```
#include "cuda/common/inc/helper_cuda.h"

using namespace std;

namespace minisam {

CudaDirectSolver::CudaDirectSolver()
    : cusolverSpH(NULL), cusparseH(NULL), stream(NULL), descrA(NULL),
      d_info(NULL), d_csrRowPtrA(NULL), d_csrColIndA(NULL),
       ↪ d_csrValA(NULL),
      d_x(NULL), d_b(NULL), d_r(NULL), h_x(NULL), is_normal(true){
}

CudaDirectSolver::CudaDirectSolver(const Eigen::SparseMatrix<double> &A,
                                   const ::Eigen::VectorXd &b,
                                   const bool normal)
    : cusolverSpH(NULL), cusparseH(NULL), stream(NULL), descrA(NULL),
      d_info(NULL), d_csrRowPtrA(NULL), d_csrColIndA(NULL),
       ↪ d_csrValA(NULL),
      d_x(NULL), d_b(NULL), d_r(NULL), h_x(NULL), is_normal(normal) {
    initialize(A, b);
}

void CudaDirectSolver::initialize(const Eigen::SparseMatrix<double>
 ↪ &A_src,
    const Eigen::VectorXd &b_src) {

#ifdef USE_DOUBLE_PRECISION
    const auto A = csc2csr(A_src);
    const auto& b = b_src;
#else
    const Eigen::SparseMatrix<float> A = csc2csr(A_src).cast<float>();
    const Eigen::VectorXf b = b_src.cast<float>();
#endif

    nnzA = A.nonZeros();           // number of nonzeros of A
    baseA = A.outerIndexPtr()[0]; // base index in CSR format

    checkCudaErrors(cudaMalloc((void **)&d_csrRowPtrA, sizeof(int) *
     ↪ (rowsA + 1)));
    checkCudaErrors(cudaMalloc((void **)&d_csrColIndA, sizeof(int) *
     ↪ nnzA));
    checkCudaErrors(cudaMalloc((void **)&d_csrValA, sizeof(TYPE) *
     ↪ nnzA));
    checkCudaErrors(cudaMalloc((void **)&d_x, sizeof(TYPE) * colsA));
    checkCudaErrors(cudaMalloc((void **)&d_b, sizeof(TYPE) * rowsA));
    checkCudaErrors(cudaMalloc((void **)&d_r, sizeof(TYPE) * rowsA));

    // note that Eigen uses CSC format, while cusparse uses CSR format
    // but here since A is symetric, it does not matter in this case
    checkCudaErrors(cudaMemcpy(d_csrValA, A.valuePtr(), nnzA *
     ↪ sizeof(TYPE), cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_csrRowPtrA, A.outerIndexPtr(), (1 +
     ↪  A.outerSize()) * sizeof(int), cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_csrColIndA, A.innerIndexPtr(), nnzA *
     ↪  sizeof(int), cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_b, b.data(), sizeof(TYPE) * rowsA,
     ↪  cudaMemcpyHostToDevice));
    h_x = (TYPE *)malloc(sizeof(TYPE) * colsA);

    checkCudaErrors(cusolverSpCreate(&cusolverSpH));
    checkCudaErrors(cusparseCreate(&cusparseH));
    checkCudaErrors(cudaStreamCreate(&stream));
    checkCudaErrors(cusolverSpSetStream(cusolverSpH, stream));
    checkCudaErrors(cusparseSetStream(cusparseH, stream));
    checkCudaErrors(cusparseCreateMatDescr(&descrA));
    checkCudaErrors(cusparseSetMatType(descrA,
     ↪  CUSPARSE_MATRIX_TYPE_GENERAL));

    if (baseA)
        checkCudaErrors(cusparseSetMatIndexBase(descrA,
         ↪  CUSPARSE_INDEX_BASE_ONE));
    else
        checkCudaErrors(cusparseSetMatIndexBase(descrA,
         ↪  CUSPARSE_INDEX_BASE_ZERO));
}


void CudaDirectSolver::solve_cholesky(Eigen::VectorXd& x) {
    // the constant used in cusolverSp
    // singularity is -1 if A is invertible under tol
    // tol determines the condition of singularity
    int singularity = 0;
    const double tol = 1.e-14;

#ifdef USE_DOUBLE_PRECISION
    checkCudaErrors(cusolverSpDcsrlsvchol(
            cusolverSpH, rowsA, nnzA, descrA,
            d_csrValA, d_csrRowPtrA, d_csrColIndA,
            d_b, tol, CHOLESKY_SYMRCM, d_x, &singularity));
#else
    checkCudaErrors(cusolverSpScsrlsvchol(
        cusolverSpH, rowsA, nnzA, descrA,
        d_csrValA, d_csrRowPtrA, d_csrColIndA,
        d_b, tol, CHOLESKY_SYMRCM, d_x, &singularity));
#endif

    checkCudaErrors(cudaMemcpy(h_x, d_x, sizeof(TYPE)*colsA,
     ↪  cudaMemcpyDeviceToHost));
#ifdef USE_DOUBLE_PRECISION
    x = Eigen::Map<Eigen::VectorXd> (h_x, colsA);
#else
    x = Eigen::Map<Eigen::VectorXf>(h_x, colsA).cast<double>();
#endif
```

```
}

void CudaDirectSolver::solve_spQR(Eigen::VectorXd& x){

    csrqrInfo_t d_info = NULL;
    size_t size_internal = 0;
    size_t size_chol = 0;
    void *buffer_gpu = NULL;

    const double zero = 0.0;

    int singularity = 0;
    const double tol = 1.e-14;

    checkCudaErrors(cusolverSpCreateCsrqrInfo(&d_info));

    checkCudaErrors(cusolverSpXcsrqrAnalysis(
        cusolverSpH, rowsA, colsA, nnzA,
        descrA, d_csrRowPtrA, d_csrColIndA,
        d_info));

#ifdef USE_DOUBLE_PRECISION
    checkCudaErrors(cusolverSpDcsrqrBufferInfo(
        cusolverSpH, rowsA, colsA, nnzA,
        descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
        d_info,
        &size_internal,
        &size_chol));

    checkCudaErrors(cudaMalloc(&buffer_gpu, sizeof(char) * size_chol));

    checkCudaErrors(cusolverSpDcsrqrSetup(
        cusolverSpH, rowsA, colsA, nnzA,
        descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
        zero,
        d_info));

    checkCudaErrors(cusolverSpDcsrqrFactor(
        cusolverSpH, rowsA, colsA, nnzA,
        NULL, NULL,
        d_info,
        buffer_gpu));

    checkCudaErrors(cusolverSpDcsrqrSolve(
        cusolverSpH, rowsA, colsA, d_b, d_x, d_info, buffer_gpu));
#else
    checkCudaErrors(cusolverSpScsrqrBufferInfo(
        cusolverSpH, rowsA, colsA, nnzA,
        descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
        d_info,
        &size_internal,
        &size_chol));
    checkCudaErrors(cudaMalloc(&buffer_gpu, sizeof(char) * size_chol));

    checkCudaErrors(cusolverSpScsrqrSetup(
        cusolverSpH, rowsA, colsA, nnzA,
        descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
        zero,
        d_info));

    checkCudaErrors(cusolverSpScsrqrFactor(
        cusolverSpH, rowsA, colsA, nnzA,
        NULL, NULL,
        d_info,
        buffer_gpu));

    checkCudaErrors(cusolverSpScsrqrSolve(
        cusolverSpH, rowsA, colsA, d_b, d_x, d_info, buffer_gpu));
#endif

    checkCudaErrors(cudaMemcpy(h_x, d_x, sizeof(TYPE)*colsA,
     ↪  cudaMemcpyDeviceToHost));
#ifdef USE_DOUBLE_PRECISION
    x = Eigen::Map<Eigen::VectorXd>(h_x, colsA);
#else
    x = Eigen::Map<Eigen::VectorXf>(h_x, colsA).cast<double>();
#endif

    if (buffer_gpu)
        checkCudaErrors(cudaFree(buffer_gpu));
    if (d_info)
        checkCudaErrors(cusolverSpDestroyCsrqrInfo(d_info));
}

void CudaDirectSolver::free_all() {
    if (h_x)
        free(h_x);

    if (cusolverSpH)
        checkCudaErrors(cusolverSpDestroy(cusolverSpH));
    if (cusparseH)
        checkCudaErrors(cusparseDestroy(cusparseH));
    if (stream)
        checkCudaErrors(cudaStreamDestroy(stream));
    if (descrA)
        checkCudaErrors(cusparseDestroyMatDescr(descrA));

    if (d_csrValA)
        checkCudaErrors(cudaFree(d_csrValA));
    if (d_csrRowPtrA)
        checkCudaErrors(cudaFree(d_csrRowPtrA));
    if (d_csrColIndA)
        checkCudaErrors(cudaFree(d_csrColIndA));
    if (d_x)
        checkCudaErrors(cudaFree(d_x));
    if (d_b)
        checkCudaErrors(cudaFree(d_b));
    if (d_r)
        checkCudaErrors(cudaFree(d_r));
}

}
```

- minisam/linear/DirectSolvers.h

```
#pragma once

#include <Eigen/Core>
#include <Eigen/SparseCore>

namespace minisam {

/**
 * wrapper of Eigen SparseQR, using built-in COLAMD ordering
 * @param LHS matrix
 * @param RHS vector
 * @param output x
 */
void qrSolver(const Eigen::SparseMatrix<double>& A, const
 ↪  Eigen::VectorXd& b,
    Eigen::VectorXd& x);

/**
 * wrapper of Eigen SparseLDLT, using built-in COLAMD ordering
 * @param LHS matrix
 * @param RHS vector
 * @param output x
 */
void choleskySolver(const Eigen::SparseMatrix<double>& A, const
 ↪  Eigen::VectorXd& b,
    Eigen::VectorXd& x);

/**
 * wrapper of Eigen Conjugate Gradient Method (Symmetric AtA)
 * @param LHS matrix
 * @param RHS vector
 * @param output x
 */
void conjugateGradientSolver(const Eigen::SparseMatrix<double>& A, const
 ↪  Eigen::VectorXd& b,
    Eigen::VectorXd& x);

/**
 * wrapper of Eigen Conjugate Gradient Method for rectangular
 ↪  least-square problem
 * @param LHS matrix
 * @param RHS vector
 * @param output x
 */
void conjugateGradient_leastSquareSolver(const
 ↪  Eigen::SparseMatrix<double> &A, const Eigen::VectorXd &b,
                                         Eigen::VectorXd &x);

/**
 * wrapper of Cuda Sparse Cholesky solver (from cuSolver)
 * @param LHS matrix
 * @param RHS vector
 * @param output x
 */
void cuda_Cholesky(const Eigen::SparseMatrix<double> &A, const
 ↪  Eigen::VectorXd &b, Eigen::VectorXd &x);

/**
 * wrapper of Cuda Sparse QR solver (from cuSolver)
 * @param LHS matrix
 * @param RHS vector
 * @param output x
 */
void cuda_QR(const Eigen::SparseMatrix<double> &A, const Eigen::VectorXd
 ↪  &b,
                   Eigen::VectorXd &x);

}
```

- minisam/linear/DirectSolvers.cpp

```
#include <minisam/linear/DirectSolver.h>
#include <minisam/linear/cuDirectSolver.h>
#include <Eigen/SparseQR>
#include <Eigen/SparseCholesky>
#include <Eigen/IterativeLinearSolvers>

#include <iostream>

using namespace std;

namespace minisam {

void qrSolver(const Eigen::SparseMatrix<double>& A, const
 ↪  Eigen::VectorXd& b,
  Eigen::VectorXd& x) {

  cout << "Solving Least Square with sparse QR Solver " <<
      "Matrix A size (MxN): " << A.rows() << "x" << A.cols() << endl;

  Eigen::SparseQR<Eigen::SparseMatrix<double>,
   ↪  Eigen::COLAMDOrdering<int>> qr(A);
  if (qr.info() != Eigen::Success)
    throw std::runtime_error("[qrSolver] QR error");
  x = qr.solve(b);
}
```

```
void choleskySolver(const Eigen::SparseMatrix<double>& A, const
 ↪  Eigen::VectorXd& b,
    Eigen::VectorXd& x) {

    cout << "Solving Least Square with Cholesky Solver. " <<
      "Matrix AtA size (NxN): " << A.cols() << "x" << A.cols() << endl;

  // prepare AtA and Atb
  Eigen::SparseMatrix<double> AtA(A.cols(), A.cols());
  AtA.selfadjointView<Eigen::Lower>().rankUpdate(A.adjoint());
  Eigen::VectorXd Atb = A.adjoint() * b;

  Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>, Eigen::Lower,
      Eigen::COLAMDOrdering<int>> chol(AtA);
  if (chol.info() != Eigen::Success)
    throw std::runtime_error("[choleskySolver] Cholesky error");
  x = chol.solve(Atb);
}

void conjugateGradientSolver(const Eigen::SparseMatrix<double>& A,
  const Eigen::VectorXd& b, Eigen::VectorXd& x) {
    cout << "Solving Least Square with (Incomplete Cholesky)
     ↪  Preconditioned Conjugate Gradient Solver" <<
      "Matrix AtA size (NxN): " << A.cols() << "x" << A.cols() << endl;

    Eigen::SparseMatrix<double> AtA(A.cols(), A.cols());
    AtA = A.transpose() * A;
    Eigen::VectorXd Atb = A.transpose() * b;

    Eigen::ConjugateGradient<Eigen::SparseMatrix<double>, Eigen::Upper,
     ↪  Eigen::IncompleteCholesky<double> > cg;

    cg.compute(AtA);
    x = cg.solve(Atb);
  }

void conjugateGradient_leastSquareSolver(const
 ↪  Eigen::SparseMatrix<double>& A, const Eigen::VectorXd& b,
    Eigen::VectorXd& x) {
    cout << "Solving Least Square with Conjugate Gradient Solver on
     ↪  rectangular matrix. " <<
      "Matrix A size (MxN): " << A.rows() << "x" << A.cols() << endl;

    Eigen::LeastSquaresConjugateGradient<Eigen::SparseMatrix<double> >
     ↪  cg;

    cg.compute(A);
    x = cg.solve(b);
  }

void cuda_QR(const Eigen::SparseMatrix<double> &A, const Eigen::VectorXd
 ↪  &b,
             Eigen::VectorXd &x) {
  CudaDirectSolver cusolver(A, b, false);
  cusolver.solve_spQR(x);
}

void cuda_Cholesky(const Eigen::SparseMatrix<double> &A, const
 ↪  Eigen::VectorXd &b, Eigen::VectorXd &x) {
  // prepare AtA and Atb
  cout << "Solving Least Square with cuSolver Cholesky "
       << "Matrix AtA size (NxN): " << A.cols() << "x" << A.cols() <<
        ↪  endl;

  Eigen::SparseMatrix<double> AtA(A.cols(), A.cols());
  AtA = A.transpose() * A;
  Eigen::VectorXd Atb = A.transpose() * b;

  CudaDirectSolver cusolver(AtA, Atb);
  cusolver.solve_cholesky(x);
}

}
```