GPU Accelerated Sequential Quadratic Programming

Xiukun Hu
University of Wyoming
Department of Mathematics
Laramie, WY 82071-3036, USA
xhu4@uwyo.edu

Craig C. Douglas, Ph.D., IEEE Member
University of Wyoming
School of Energy Resources and Department of Mathematics
Laramie, WY 82071-3036, USA
craig.c.douglas@gmail.com

Robert Lumley and Mookwon Seo
AirLoom Energy, LLC
Laramie, WY 82072
robert.lumley@airloomenergy.com and mookwon.seo@airloomenergy.com

***Abstract*—Nonlinear optimization problems arise in all industries. Accelerating optimization solvers is desirable. Efforts have been made to accelerate interior point methods for large scale problems. However, since the interior point algorithm used requires many function evaluations, the acceleration of the algorithm becomes less beneficial. We introduce a way to accelerate the sequential quadratic programming method, which is characterized by minimizing function evaluations, on graphical processing units.**

Keywords-nonlinear optimization; SQP; GPU; CUDA; linear solvers

I. INTRODUCTION

An optimization problem is a problem that can be reformed as

$$\min_{\boldsymbol{x}} f(\boldsymbol{x})$$

such that

$$\boldsymbol{g}(\boldsymbol{x}) \geq 0 \text{ and}$$

$$\boldsymbol{h}(\boldsymbol{x}) = 0,$$

where $\boldsymbol{x}$ is the vector of variables, $f(\boldsymbol{x})$ is the objective function, and $\boldsymbol{g}(\boldsymbol{x})$ and $\boldsymbol{h}(\boldsymbol{x})$ are inequality and equality constraints, respectively.

When we are trying to find a best solution, we meet an optimization problem. Companies in different industries solve huge and complicated nonlinear nonconvex optimization problems to make decisions. As the growth of computational power and the development of optimization algorithms, more and more variables become possible to be considered, and the relations between objective and variables, and between constraints and variables, becomes increasingly complex.

Sequential quadratic programming (SQP) methods are iterative nonlinear optimization methods that solves a sequence of quadratic programming (QP) subproblems to solve the problem. They are preferable when the evaluation of any of f, $\boldsymbol{g}$, or $\boldsymbol{h}$ is expensive. In contrast to other nonlinear optimization methods, the SQP method places more effort in path finding in order to minimize function evaluation time. Thanks to this feature, the whole overall application can enjoy the speed up provided by the SQP method. For other optimization methods, e.g., interior point methods, which spent most of the time evaluating functions, the acceleration of the method itself usually makes little difference [1].

In this paper, we introduce an SQP algorithm based on Wright [2] and Mehrotra [3], and describe our method for accelerating the SQP with CUDA's cuSolver library. In Section II, we give a brief introduction to the cuSolver library, a high level direct linear solver package. We also list the limiting factors of parallel sparse solver. In Section III, we give a short description of the specific SQP algorithm we are trying to accelerate, including a line search SQP algorithm and a predictor-corrector QP algorithm. We observe the possible way to take advantage of cuSolver. In Section IV, we describe the modifications and adjustments we made to achieve a good performance. In Section V, we compare our code with KNITRO's SQP algorithm [15]. In Section VI, we draw some conclusions and discuss future work.

II. THE CUSOLVER LIBRARY

Starting in CUDA 7, NVIDIA expanded its capabilities of computation with a direct linear solver library, cuSolver [4]. This library provides LAPACK-like features including matrix factorization, triangular solve routines for dense matrices, a sparse least-squares solver, and an eigenvalue solver. It has three major components: cuSolverDN, cuSolverSP, and cuSolverRF:

- cuSolverDN provides dense matrix factorization and solve routines such as LU, QR, SVD and LDLT.
- cuSolverSP provides sparse routines based on a sparse QR factorization.
- cuSolverRF is a sparse refactorization package for solving a sequence of matrices with the same sparsity pattern. Only the LU factorization method is provided.

In our case, as we show in Section III, we need to solve a sequence of symmetric, indefinite sparse matrices, which

2473-3636/17 $31.00 © 2017 IEEE
DOI 10.1109/DCABES.2017.8

can be numerically singular. Thus, we focus on the sparse QR solver in cuSolverSP.

There are three different QR solver APIs in cuSolverSP:

- csrlsqvqr() is the basic QR solver. It solves one single least square problem with one compressed sparse row format (CSR) coefficient matrix at a time.
- csrqrBatched() solves a set of least square problems to achieve higher concurrency. It requires all of the matrices in each batch share to have the same sparsity pattern.
- csrqrFactor() and csrqrSolve() is a pair of low level functions that do factorization and solving separately. This is useful when the coefficient matrix of a least square system is used for multiple right hand sides, where these right hand sides cannot be provided at once.

To achieve the best performance, we need to understand the factors that affect the process. Solving a sparse linear system in parallel needs consideration of multiple factors.

A. *Fill-In.*

The "fill-in" are those entries in the sparse matrix that change from an zero to a nonzero value during factorization. A large number of fill-ins ruins the performance. To take advantage of the sparsity, fill-in of the sparse matrix must be avoided. Reordering the matrix by using the approximate minimum degree (AMD) algorithm or the reverse Cuthill-McKee (RMC) ordering can dramatically decrease the fill-in, computation, and memory required.

B. *Concurrency*

GPUs require a considerable concurrency to achieve satisfying performance. A single matrix, especially a small or medium one, usually fails to have enough concurrency to make using a GPU preferable. CUDA provides batched linear solver approaches. These approaches can solve multiple linear systems with the same sparsity patterns at the same time (e.g., using the csrqrBatched() function mentioned above). This approach can dramatically increase the concurrency and achieve far higher occupancy on GPU.

C. *The Memory Access Pattern*

In GPU computing, the memory access pattern decides the memory efficiency. Since the numerical factorization has no regular access pattern, this becomes the bottleneck for GPU direct solvers.

III. PROGRAMMING ALGORITHM

A. *A Line Search Algorithm for SQP*

The basic idea of this SQP method is, in each iteration, to approximate the original problem at a given trial point $\boldsymbol{x}_k$ using a quadratic programming subproblem. The solution of this subproblem then becomes the search direction and determines a new trial point $\boldsymbol{x}_{k+1}$ for the next iteration. A line search algorithm is applied to find $\boldsymbol{x}_{k+1}$, and a new quadratic subproblem is then constructed. The sequence of these trial points will converge to a local minimum if the problem is both feasible and well bounded.

In each iteration of the SQP algorithm, a QP subproblem is defined by

$$\min_p F(\boldsymbol{p}) = \nabla \boldsymbol{f}^{\mathrm{T}} \boldsymbol{p} + 1/2\, \boldsymbol{p}^{\mathrm{T}} \boldsymbol{B}\, \boldsymbol{p},$$

subject to

$$\boldsymbol{J_g p} + \boldsymbol{g} \geq 0, \text{ and}$$

$$\boldsymbol{J_h p} + \boldsymbol{h} = 0,$$

where $\nabla \boldsymbol{f}$ is the gradient vector of the objective function f, $\boldsymbol{J_g}$ and $\boldsymbol{J_h}$ are Jacobian matrices of $\boldsymbol{g}$ and $\boldsymbol{h}$ at $\boldsymbol{x}_k$, respectively, and $\boldsymbol{B}$ approximates the Hessian matrix of f and is usually computed by a Quasi-Newton methods. $\boldsymbol{B}$ can also be the exact Hessian if the evaluation of the Hessian is provided and inexpensive. In our codes, we use the damped Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm to approximate $\boldsymbol{B}$ in each iteration.

Even though multiple line search algorithms can be applied for each subproblem, the time to solve all of the quadratic subproblems always consumes most of the solve time.

B. *A Predictor-Corrector Algorithm for QP*

A quadratic programming problem is a problem with a form

$$\min_x q(\boldsymbol{x}) = 1/2\, \boldsymbol{x}^{\mathrm{T}}\, \boldsymbol{G}\, \boldsymbol{x} + \boldsymbol{x}^{\mathrm{T}}\, \boldsymbol{c},$$

subject to

$$\boldsymbol{A_i}\, x + \boldsymbol{b}_i \geq 0, \text{ and}$$

$$\boldsymbol{A_e x} + \boldsymbol{b}_e = 0.$$

We use the predictor-corrector QP method introduced by Mehrotra [3]. This algorithm solves the problem by iteratively solving the primal-dual Karush-Kuhn-Tucker (KKT) linear system with a coefficient matrix with the form

$$\begin{pmatrix} \boldsymbol{G} & \boldsymbol{0} & -\boldsymbol{A_e}^{\boldsymbol{T}} & -\boldsymbol{A_i}^{\boldsymbol{T}} \\ \boldsymbol{A_e} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{A_i} & -\boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Z} & \boldsymbol{0} & \boldsymbol{S} \end{pmatrix}$$

where $\boldsymbol{I}$ is the identity matrix and $\boldsymbol{Z}$ and $\boldsymbol{S}$ are diagonal matrices that are updated in each iteration. The algorithm iteratively updates an initial guess of the solution, $\boldsymbol{x}_k$, using Lagrange multipliers $\boldsymbol{y}$ and $\boldsymbol{z}$, by solving the above equation. Notice that in each iteration, only the Z and S parts are changed.

In our algorithm, we reordered the above coefficient matrix to gain symmetry. The reformatted matrix is below, where the Hessian approximation matrix $\boldsymbol{G}$ is always symmetric:

$$\begin{pmatrix} \boldsymbol{G} & \boldsymbol{0} & -\boldsymbol{A}_e^T & -\boldsymbol{A}_i^T \\ \boldsymbol{0} & \boldsymbol{S}^{-1}\boldsymbol{Z} & \boldsymbol{0} & \boldsymbol{I} \\ -\boldsymbol{A}_e & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{A}_i & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \end{pmatrix}$$

The whole SQP algorithm then can be regarded as a sequential sparse linear solver. Usually solving these systems accounts for more than 90% of the runtime of the overall SQP solver.

C. Concurrency Analysis

Due to the dependency between each linear system, when solving a single SQP problem, there is not much concurrency we can expect, especially for small and medium size problems, which is a common case for nonlinear optimization problems. To achieve reasonable concurrency, we must run multiple starting points.

Since most of the optimization problems in real world are not convex, convergence to a global minimum is not necessarily guaranteed by the optimization solver. To get a relatively optimal solution, running the optimization solver with as many as possible different initial points and find the best answer of all these local optimum is a trivial but common method, usually named as multiple-start algorithm.

Recall from Section II that the cuSolver batched approach requires all linear systems to share the same sparsity pattern. Section IV illustrates the method we used. It also documents some ogher efforts we tried to exploit more of the computational power of a GPU.

IV. GPU Accelerated SQP

According to the SQP algorithm introduced in Section III, the sparsity pattern of the linear system in each QP subproblem remains untouched. Further, by equation (3), each matrix is constructed from the Hessian approximation matrix $\boldsymbol{B}$ and Jacobian matrices $\boldsymbol{J}_g$ and $\boldsymbol{J}_h$. If we fix the sparsity pattern of these three matrices throughout the entire SQP solution process, then all the matrices we need to solve for a specific optimization problem will always have the same sparsity pattern.

The algorithm can be described as follows.

1. Initialize m optimization problems on the CPU, collect all linear systems, and upload to GPU.
2. Do symbolic analysis and calculate the required buffer size on GPU. Then allocate the buffer.
3. Launch the cuSolver API to solve the batched linear systems using QR decomposition.
4. Copy back the solutions from the GPU to the CPU, and generate the next m linear systems based on these solutions on CPU. Go back to step 3.

Before further optimization, the algorithm turns out to be much slower on the GPU than a high performance CPU only SQP solver.

Table I shows the performance we achieved before further optimization. The optimization problem is the test problem *largest small polygon* from COPS 3.0 [5]. The problem size has 38 variables and 265 inequality constraints (i.e., it is a polygon with 20 vertices). Now we give two main reasons why the GPU solver runs about 3.5 times slower than the CPU code.

TABLE I. GPU SQP Performance Before Optimization

Solver	Time	Objective (max)
GPU single precision	2.13s	0.7762
GPU double precision	4.28s	0.7768
KNITRO 10.3 SQP	1.21s	0.7758

The first reason lies in fixing sparsity pattern. We fixed the sparsity pattern by padding zeros into $\boldsymbol{B}$, $\boldsymbol{J}_g$ and $\boldsymbol{J}_f$. Since we are using a finite difference method to approximate $\boldsymbol{J}_g$ and $\boldsymbol{J}_f$, we can only assume that they are both dense. Hence, we unfortunately added considerable nonzeroes into our linear system, since in most optimization problems, f, g and h will not be related to all variables. This slows down the whole optimization program, compared to when the sparsity pattern can be varied and all zeros can be ignored. We solved this problem by supplying exact functions for evaluating the Jacobian matrices. This fixes the sparsity pattern problem and minimizes the density. After this modification, the whole program gains more than a three times speed up.

The second reason is the fill-in. As illustrated in Section II, fill-in can dramatically slow down the solver. We then use the AMD algorithm from the csparse library [6] along with cuSolver permutation functions to reorder our matrices and minimize fill-in before solving them. Because all of the matrices have the same sparsity pattern, this preordering method need only to be applied to a single matrix. The resulting permutation can be used in every linear solving process. After this modification, our code achieves about a 20 times speed up for both single and double precision.

We also use multiple streams of batched solvers on a single CPU thread to run alternately, in order to hide the memory transfer time between the CPU and the GPU.

V. Performance

We tested the performance of our GPU accelerated SQP method on the *largest small polygon* from COPS 3.0. The result is compared with KNITRO 10.3. The GPU used is a GTX 1080 and the CPU is an Intel Core i7-6950X. The CPU solver is KNITRO 10.3.0 written in C++ and the algorithm is set to SQP. The GPU accelerated code is based on CUDA toolkit 9.0 RC. With the number of vertices equal to n_v, the number of variables is $2n_v - 2$, and the number of inequality constraints has the order of n_v^2. There is no equality constraint. For details please refer to Dolan [5].

Table II shows the average time spent for a single start point using our GPU accelerated SQP method with different batch sizes. It runs in single precision with n = 50. It can be seen that the batched approach provides tremendous performance.

TABLE II. GPU SQP Performance Versus Batch Size

Batch size	GPU time
1	13.2375
2	6.1375
4	2.89375
8	1.371875
16	0.7321875
32	0.407265625
64	0.279453125
128	0.314609375
256	0.315695313

Table III compares our GPU SQP algorithm in double precision with the KNITRO SQP algorithm [15], where n is the number of vertices. It shows that our algorithm runs 6-20 times faster than the KNITRO SQP code. We comment that our solutions are also better than those produced by the KNITRO SQP code.

TABLE III. GPU SQP VERSUS KNITRO

n	GPU time (s)	KNITRO time (s)	GPU objective	KNITRO objective
10	0.015	0.133	0.7491	0.7491
20	0.054	1.209	0.7768	0.7758
30	0.466	2.549	0.7810	0.7810
40	1.680	10.132	0.7832	0.7832

VI. CONCLUSION AND FUTURE WORK

The acceleration of the SQP method can benefit many optimization problems. The key is to find a fast sparse direct solver. However, because of insufficient parallelism, random access patterns, and fill-in of the sparsity pattern, it becomes quite a challenge. We illustrated an approach to gain concurrency by using multiple start points at the same time and forcing the sparsity pattern to be the same. We also demonstrated that the preordering is essential to achieving performance using cuSolver. By using the AMD ordering algorithm, we finally achieved a high quality acceleration.

There is still a lot of room for improvement. First, there are two linear system in each iteration of the QP subproblem with the same coefficient matrix. If we save the factorization information for reuse in the second solve, the program can potentially run about twice faster. Second, we tested different sparse direct solvers on the CPU with Matlab. The LDL^T decomposition method (using MA57 algorithm, see Duff [7]) works much faster than other solvers on our test problems. If we can implement the multifrontal LDLT method on a GPU, then a potential huge speed up can be expected.

ACKNOWLEDGMENT

This research originated as a project at Airloom Energy, LLC that was sponsored by University of Wyoming and supported in part by the National Science Foundation.

REFERENCE

[1] Y. Cao, A. Seth, and C. D. Laird, "An augmented Lagrangian interior-point approach for large-scale NLP problems on graphics processing units," Computers & Chemical Engineering, Vol. 85, 2 February 2016, pp. 76-83.

[2] J. Nocedal and S. J. Wright, Numerical Optimization, 2nd ed., Springer New York, 2006.

[3] S. Mehrotra, "On the Implementation of a Primal-Dual Interior Point Method," SIAM Journal on Optimization, Vol. 2, 1992, pp. 575-601.

[4] Cuda C. Programming guide, 2012.

[5] E. D. Dolan, J. J. Mor´e and T. S. Munson, "Benchmarking optimization software with COPS 3.0," Mathematics and Computer Science Division, Argonne National Laboratory, Technical Report ANL/MCS-273, February 2004.

[6] T. Davis, Direct Methods for Sparse Linear Systems, SIAM, 2006.

[7] I. S. Duff, "MA57 - a code for the solution of sparse symmetric definite and indefinite systems," ACM Trans. Math. Softw. 30, 2, June 2004, pp. 118-144.

[8] T. R. Kruth, "Interior-point algorithms for quadratic programming," Master's thesis, Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2008.

[9] A. George and J. W. Liu, "The evolution of the minimum degree ordering algorithm," Siam review, 31(1), 1989, pp.1-19.

[10] A. R. Conn, N. I. Gould, and P. L. Toint, "Testing a class of methods for solving minimization problems with simple bounds on the variables," Mathematics of Computation, 50(182), 1988, pp.399-430.

[11] L. Pólik and T. Terlaky, "Interior point methods for nonlinear optimization," Nonlinear optimization, Springer Berlin Heidelberg, 2010, pp. 215-276.

[12] I. S. Duff, A. M. Erisman, and J. K. Reid, Direct methods for sparse matrices, Oxford University Press, 2017.

[13] S. Axler, Linear algebra done right, 2nd ed., Springer, New York, 1997.

[14] W. H. Press, Numerical Recipes 3rd edition: The art of scientific computing, Cambridge University Press, 2007.

[15] R. H. Byrd, J. Nocedal, and R.A. Waltz, "KNITRO: An integrated package for nonlinear optimization", in G. di Pillo and M. Roma, editors, Large-Scale Nonlinear Optimization, Nonconvex optimization and its applications series, vol. 83, Springer, New York, 2006, pp. 35–59.