GPU algorithms for large-scale optimization

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Chemical Engineering

Benjamin F. Sauk

B.S., Chemical and Biomolecular Engineering, Georgia Institute of Technology

Carnegie Mellon University

Pittsburgh, Pennsylvania

July, 2020

Acknowledgments

I am extremely grateful to my academic advisor Professor Nikolaos V. Sahinidis for all of his time and support over the last five years. Nick has constantly pushed me to become a better writer, presenter, and researcher. His insightful questions have challenged me to understand the fundamental concepts of my work, and to succinctly explain any conclusion that I reach. I appreciate Nick's patience with me, and I value all of his feedback. I also thank Nick for the freedom he gave me to explore several different research directions on topics that I was passionate about. I fondly recall several Skype and Zoom calls where we spent hours improving manuscripts. The lessons I have learned from Nick will stick with me, and help me for the rest of my life. I wish him the best as he continues his adventure at my undergraduate alma mater, Georgia Tech.

I want to express my gratitude to my doctoral committee members: Professor Larry Biegler, Professor Franz Franchetti, and Professor Chrysanthos Gounaris. I appreciate all of the kindness and feedback that all three have provided during my PhD. The ideas and comments that I have received from my committee have shaped the direction of my dissertation, and led to a series of interesting discoveries. My thanks to Professor Franchetti who inspired me to investigate autotuners and supercomputers, leading me to conduct experiments on the Pittsburgh Supercomputing Center computers. I will treasure the classes that I took with Larry and Chrysanthos as they prepared me not only for my time at CMU, but also for my entire career. I also thank both of them and all of the PSE faculty at CMU.

This work was conducted as part of the Institute for the Design of Advanced Energy Systems (IDAES) with funding from the Office of Fossil Energy, Cross-Cutting Research, U.S. Department of Energy. I want to thank the entire IDAES team who have provided me with the unique opportunity to work with researchers from several institutions and national labs. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC). We also gratefully acknowledge the support of the NVIDIA Corporation with the donation of the NVIDIA Tesla K40 GPU used for this research.

I have had the privilege of working with a great group of people in the Sahinidis group. I am especially indebted to Professor Nikolaos Ploskas. Throughout my entire PhD, Nikos has been incredibly patient and kind. He has been an essential contributor to several of my projects at CMU. I am especially grateful for him helping me debug graphical interface issues associated with installing NVIDIA drivers. I also appreciate all of the former and current Sahinidis students that I encountered at CMU: Dr. Yash Puranik, Dr. Nick Austin, Dr. Sree Rajagopalan, Dr. Zach Wilson, Dr. Carlos Nohra, Dr. Tong Zhang, Marissa Engle, Brad Johnson, Yijia Sun, Christian Hubbs, Kaiwen Ma, Owais Sarwar, and Anatoliy Kuznetsov. To Tong, thank you for all of the discussions we had over the years, and for teaching me about so many things. I also am fortunate to have interacted with the entire PSE community. I enjoyed spending time and learning from the entire PSE group, and I will cherish the time I spent with all of you.

I have been extremely fortunate to have spent the last five years with an outstanding group of classmates and friends. Everyone at CMU has contributed to making my time here truly special. With special thanks to: Dr. Brittany Nordmark, Dr. Pablo García, Dr. Qin Gu, Dr. Christopher Hanselman, Dr. Yun-Ru Huang, Dr. Steven Iasella, Dr. Danish Iqbal, Dr. Michael Davidson, Dr. Dana McGuffin, Dr. Charles Sharkey, Dr. Qi Chen, Dr. Akang Wang, Kathy Fein, Marissa Engle, Bowen Huo, Kevin Tran, Scott Pedu, and the entire Taco Friday crew.

I would also like to thank Dr. Michael Davidson and Marissa Engle for getting lunch together almost every day for the last five years. From Subway Tuesdays through Taco Friday, I greatly appreciate all of the conversations we had, and I value all of the insights you both have provided about research and life. To my rock climbing, gym, and running companions: Qin Gu, Kevin Tran and Scott Pedu thank you very much. To Scott, thank you for wasting your time so I definitely do not have to.

To Marilyn, I cannot express my gratitude to you enough. Thank you for your kindness, and your patience with me. You inspire me to do my best everyday, and I appreciate all of your support and your willingness to read my papers and listen to my practice presentations. Finally, a thanks to my family for their love and support over the years. Both of my parents have instilled in me a love for learning and have supported me throughout my entire life. Thanks to Greg, Renae, and Geoff.

Benjamin Sauk
Pittsburgh, PA
July 2020

Abstract

Data-driven modeling and optimization are both core elements in process systems engineering. When first-principle models are unavailable, empirical models can serve as a surrogate for the underlying physics of a process. The advent of sophisticated sensors and increased digital storage capabilities presents the chemical industry with vast quantities of data and the opportunity to generate more realistic surrogate models. Despite the need to scale up data-driven modeling algorithms, there has been limited work in developing parallel computing techniques for model building. The goals of this dissertation are to develop efficient parallel algorithms for model building, and investigate parallel approaches for optimization of linear programming problems.

In Chapter 2, we review techniques used for generating linear models and discuss using QR factorization to generate empirical models from tall and skinny matrices. We describe high performance parallel implementations for the linear least squares problem solved with QR factorization. We discover that state-of-the-art algorithms are not optimized for tall and skinny linear least squares problems. We propose the use of derivative-free optimization and simulation optimization to optimize the performance of QR factorization routines in the MAGMA [93] library. Results demonstrate that the performance of solving the linear least squares problems can be accelerated with the use of optimal tuning parameters identified by derivative-free optimization.

In Chapter 3, we revisit the problem of algorithmic parameter tuning. We review the literature on auto-tuning techniques and propose a hybrid derivative-free optimization strategy for auto-tuning. Our approach combines local and global derivative-free optimization with

the multi-armed bandit function. We implement our hybrid algorithm in an open-source tool, HybridTuner, furthering the development of sophisticated autotuners and assisting others in optimizing code performance without exhaustive tuning.

After addressing how to efficiently generate linear models, we describe approaches for identifying a sparse set of linear regressors. In Chapter 4, we propose to solve the best subset selection problem with backward stepwise elimination. We develop a theoretical guarantee on the accuracy of backward stepwise elimination using the supermodularity ratio, and compare its efficacy against other subset selection strategies like the lasso [166], and forward stepwise selection [58].

Finally, in Chapter 5, we discuss methods for parallelizing the simplex algorithm. We consider previous methods for solving linear programming problems and parallelize the block-LU update [70] with GPU computing. We solve quantile regression linear programming problems up to a factor of 4.03x faster than a sequential primal simplex algorithm. We also demonstrate that our approach is resistant to numerical instability when solving quantile regression problems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The last decade has seen an explosion in the usage of graphics processing units (GPUs) for general purpose computing. Originally designed to render images on a computer, GPUs have been partially re-purposed for a variety of applications ranging from dense linear algebra [42, 93] to mining crypto currency. Underlying all of these applications is the need to perform single instructions multiple data (SIMD) operations. The first fully programmable GPU was released in 2001 [144], and general purpose GPU applications were being developed within the same year [108, 82]. Several years later in 2006, NVIDIA released the CUDA programming language to create a structured way for programmers to take advantage of GPU parallel computing [130].

GPUs have successfully been applied to different applications such as iterative linear algebra [59], Monte Carlo simulations [180], sparse matrix-vector multiplication [21, 120], and bioinformatics [175]. GPUs accelerate applications where the same task needs to be performed independently on multiple sets of data. The large number of independent operations in dense linear algebra kernels make them an ideal candidate for GPU computing. Parallelizing the linear least squares problem with GPU computing is one of the fastest ways to solve it. In the area of linear programming, parallel sparse linear algebra operations can result in modest speedups on some classes of linear programming problems.

Additionally, portability of GPU algorithms has remained a major challenge. Portability

refers to the ability of software to be developed on one computer or GPU for use on another system. In particular, GPU algorithms are parameterized by several GPU specific tuning parameters that impact performance. Optimal parameters are dependent on the underlying architecture of the GPU. When an algorithm developed on one GPU is used on another, performance is lower when proper parameter tuning is not considered.

This dissertation proposes developments in parallelizing the solution of the linear least squares problems (LLSPs) and linear programming (LP) problems. Despite the ever increasing availability of data, there has been limited effort to develop efficient parallel optimization algorithms to handle larger and more nuanced data sets. When parallel algorithms are developed, parameter tuning is handled with heuristic strategies that severely limit portability and their use in practice. This thesis aims to both develop GPU parallel algorithms for large-scale computing and develop novel parameter tuning techniques to improve tuning performance when applied to GPU algorithms. In this chapter, we discuss the state-of-the-art relevant to GPU parallel computing for the problems that we consider, and describe the remaining chapters in this work.

1.1.1 Linear least squares problem

As more powerful data collection techniques become available in the chemical industry, engineers have access to more process data than ever before. With larger data sets, engineers have the potential to develop models that can better capture the physics of real systems. However, developing models that represent reality with larger data sets may lead to prohibitively expensive computational times. At its core, surrogate model generation from process data is a problem of feature selection and regression. The goal is to identify a sparse set of features that accurately relate input observations to an output measurement. The set of features comprises both measurable properties and nonlinear transformations of

those features. First, we address the problem of efficiently generating a linear model given a set of features.

In this case, a model is generated by solving the linear least squares problem (LLSP) defined as

$$\min \|Ax - b\|_2 \tag{1.1}$$

where A is a matrix of size $m \times n$, x is a vector of size n, and b is a vector of size m. Here m refers to the number of observations in the matrix A that contains information about the input variables, and n refers to the number of features considered and $m \geq n$. The solution of the LLSP satisfies the *normal equation*

$$A^T Ax = A^T b. \tag{1.2}$$

The normal equation is solved with QR factorization, where A is decomposed into an orthogonal matrix Q and an upper triangular matrix R. One common element in LLSPs is that A is usually tall and skinny, i.e., $m >> n$. While parallel GPU algorithms have been developed for QR factorization, no previous work has considered investigating the portability of these methods for tall and skinny matrices. Portability is commonly addressed through algorithmic parameter tuning or autotuning. Autotuning is the problem of systematically optimizing the performance of an algorithm by adjusting tuning parameters or compiler options. Optimal tuning parameters are desirable as they can lead to significant performance benefits. For example, well-chosen parameters have led to a 25% increase in performance compared to default parameter values in the case of matrix-matrix multiplication [110].

In Chapter 2, we investigate the benefits of using derivative-free optimization (DFO) and simulation optimization (SO) to systematically optimize tunable parameters for a GPU or hybrid CPU/GPU architecture. DFO and SO address optimization problems whose objective function are not available in algebraic form. Computational experiments show

that both DFO and SO are effective tools for determining optimal tuning parameters that speed up the performance of the popular LLSP solver MAGMA by about 1.8x, compared to MAGMA's default parameters for large tall and skinny matrices. By using DFO solvers, we identify optimal parameters using an order of magnitude fewer simulations than with direct enumeration.

1.2 HybridTuner

Algorithms are commonly designed with tuning parameters. A tuning parameter can be defined as any parameter that, when modified, improves the performance of an algorithm on either a particular computer, or for a particular problem. It is nearly impossible to develop a generic algorithm that is optimal for every problem on every machine. Instead, algorithms are designed with adjustable parameters that can be modified to improve the performance of an algorithm on a particular class of problems. For example, in 2010, CPLEX 12.1 had 76 tunable parameters that adjust the performance of CPLEX for certain classes of problems. Hutter et al. demonstrated that tuning the 76 parameters led to speedups of more than a factor of 10x for certain problems [90].

As we show in Chapter 2, parameter tuning can be posed as a black-box optimization problem. Assume that there exists a relationship between algorithmic tuning parameters and an output performance metric that needs to be optimized, such as execution time, that can be obtained by querying a black-box. There may not exist an explicit algebraic functional form to relate the input tuning parameters to its output. Given that this relationship varies dynamically with the computer architecture and the problem structure, we propose to address this challenge with derivative-free optimization. As solution strategies in this field involve querying a black-box, the number of simulations required to identify optimal solutions to a problem increases with the number of variables. In practice, tuning

problems are challenging in cases with more than 30 variables, though some methods have been observed to find reasonable solutions for problems with hundreds of parameters [12]. As there is no way to validate whether optimal parameters have been found for problems with a large search space, many users determine tuning parameters with heuristics, often producing inefficient algorithms.

In Chapter 3, we propose two hybrid derivative-free optimization methods to maximize the performance of an algorithm after evaluating a small number of possible algorithmic configurations. Our autotuner (a) reduces the execution time of dense matrix multiplication by a factor of 1.4x compared to state-of-the-art autotuners, (b) identifies high-quality tuning parameters within only 5% of the computational effort required by other autotuners and (c) can be applied to any computer architecture. We demonstrate our approach for problems with up to 50 hyperparameters.

1.3 Backward stepwise elimination

Next we address the challenge of identifying a succinct, yet accurate set of features that relate input to output measurements. As datasets and models become more complex, the ordinary least squares method may lead to over-fitting models without providing model interpretability. To prevent over-fitting, a model can be created by selecting a subset of regressors. We address the feature selection problem with best subset selection.

Best subset selection is NP-hard to solve and is expensive to solve for problems with a large number of features. Many researchers have addressed the problem of sparse linear regression with exact solution techniques [76, 44, 24], L1 penalty methods [166, 119], exhaustive search, or heuristic strategies [58]. In practice, heuristic strategies are often used even though they rarely come with performance guarantees. Inspired by the work of Nemhauser et al. [126], the concept of submodular set functions can provide a lower bound when maximizing a

set function with the greedy selection algorithm subject to a cardinality constraint. There has investigated providing an approximation guarantee for the forward stepwise selection algorithm using approximate submodularity [52]. Almost no work has been done to provide an approximation guarantee on the accuracy of backward stepwise elimination.

Another challenge with subset selection is that even when it is possible to solve exactly, the mathematically optimal model may not be the best choice in practice. The work of Hastie et al. [83] demonstrates that for most problems the performance of a stepwise selection heuristic, forward stepwise selection, performed as well as models obtained by L1-regularization or models obtained by solving an integer programming formulation of subset selection with a time limit.

In Chapter 4, we investigate solving the best subset selection problem with backward stepwise elimination (BSE). We prove an approximation guarantee for BSE that bounds its performance by applying the concept of approximate supermodularity. We develop a GPU parallel BSE algorithm that is 5x faster than an efficient CPU implementation. Finally, we demonstrate the performance of BSE against several state-of-the-art feature selection approaches. For certain classes of problems, BSE generates solutions with lower relative test error than the lasso, the relaxed lasso, and forward stepwise selection.

1.4 GPU block-LU update

The simplex algorithm for solving linear programming (LP) problems is regarded as one of the most influential algorithms ever developed. LP problems allow modelers to explicitly represent the physics of a system, physical constraints, capital costs, operating costs, and determine an optimal set of control actions that maximize or minimize some objective. The simplex algorithm is central to identifying optimal solutions for linear, nonlinear, mixed-integer, and stochastic optimization problems. When solving larger and more complex

problems, efficient implementations are essential to identify accurate solutions for large industrial applications in a reasonable amount of time. Over the last 70 years, advances in both algorithms and hardware have resulted in tremendous algorithmic speedups. Parallel computing has accelerated individual simplex iterations, and across several iterations [89].

The simplex algorithm is an iterative method, in the case of the primal simplex, moving from one basic feasible solution to another until proving optimality. LU factorization and basis updates speedup the transition from one iteration to the next. Despite the use of parallel computing in several steps of the simplex algorithm, no work has been done to parallelize a method known as the block-LU update [70] with GPU computing.

In Chapter 5, we parallelize the block-LU update using the many-core processing power of GPUs. We present the benefit of this approach by solving quantile regression problems formulated as LP problems. While linear regression is obtained through minimizing the sum of squared residuals, quantile regression is the solution to minimizing the sum of absolute residuals [103]. Quantile regression is a robust estimator that outperforms linear regression when the noise is heteroskedastic, and is more robust when generating models from data with outliers [102]. A quantile regression model is obtained from solving the following LP problem:

$$\min_{(\beta,u,v)\in\mathbb{R}^r\times\mathbb{R}^{2s}_+}\{\tau 1_r^T u + (1-\tau)1_r^T v | X\beta + u - v = b\}, \tag{1.3}$$

where $\tau \in \{0,1\}$ is the desired quantile, $X \in \mathbb{R}^{r\times s}$ is a design matrix used for generating a regression model, r is the number of observations, s is the number of design variables, $b \in \mathbb{R}^r$ is the observed output response, and $1_r \in \mathbb{R}^r$ where all values are 1. The regression coefficients are stored in β, while u and v are non-negative slack variables.

We propose a hybrid CPU-GPU primal simplex algorithm that updates the basis matrix and solves linear systems of equations on a graphics processing unit (GPU). We accelerate the performance of a primal simplex algorithm with our parallel block-LU update compared

to the HSL LA04 [75] simplex algorithm for quantile regression problems. Computational results show a geometric average speedup of 1.63x for tall and skinny quantile regression problems with $\{2000, 2500, 3000\}$ rows and 1500 columns. We also demonstrate that the block-LU update is insensitive to numerical instability that results from solving quantile regression problems.

Chapter 2

LLSP Parameter Tuning

2.1 Introduction

Numerical solvers on graphics processing units (GPUs) are typically designed for one particular GPU architecture [115], and may be suboptimal or even unusable on another one [95]. Programmers have circumvented this problem by introducing tunable parameters into their algorithms that can be easily modified when the solver is being implemented on a different GPU architecture than it had been designed for [1]. However, determining tuning parameters that maximize solver performance is a challenging optimization problem because there is no explicit relationship to model the interactions between the software, algorithms, and hardware. Derivative-free optimization (DFO) [147] and simulation optimization (SO) [8] can be used to solve this problem since they do not require explicit functional representations of the objective function or of the constraints. Instead, the solver can be treated as a black-box system that accepts tuning parameters, and outputs a performance metric such as execution time or floating point operations per second (FLOPs).

This paper addresses the problem of using DFO and SO to determine optimal tuning parameters for solving linear least squares problems (LLSPs) with GPUs. Dense LLSPs are solved in a wide range of fields, such as curve fitting, modeling of noisy data, signal processing, parameter estimation, and machine learning, including best subset selection and the lasso [34, 54]. LLSPs arise when solving an overdetermined system of equations $Ax = b$, where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $x \in \mathbb{R}^n$. In modeling an input-output system, the A matrix

contains information about input variables. One common element in LLSPs is that A is usually tall and skinny (TS), i.e., $m >> n$. The vector x denotes the LLSP solution, and b is the vector of output measurements. In an overdetermined system ($m > n$), there may not exist an exact solution. The LLSP finds x that minimizes the difference between b and Ax or, more formally, $\min \|Ax - b\|_2$. It can be shown that the optimal LLSP solution satisfies the *normal equation* $A^T Ax = A^T b$.

Even though many techniques exist to solve LLSPs, we decided to use QR factorization. Given A, QR factorization determines matrices Q and R, where Q is orthogonal and R is upper triangular, such that $A = QR$. The LLSP solution with QR factorization simplifies to $x = R^{-1}Q^T b$. Other methods such as the normal equation method, and singular value decomposition, were ruled out for issues related to the numerical stability of ill-conditioned matrices [73], or more expensive computational costs, respectively [54]. Another technique, the seminormal equation method, combines the normal equation approach with QR factorization. In this method, QR factorization is used on one side of the normal equation to simplify the solution x to $R^T Rx = A^T b$ where no information is needed about Q [114]. Although this method reduces the amount of memory needed to decompose a matrix, if A is ill-conditioned, the method may not find an accurate solution.

Dense linear algebra problems were one of the earliest applications for general purpose GPU computing dating back to the early 2000's [132]. Now, NVIDIA has standard GPU libraries for both basic linear algebra subprograms (BLAS) and dense linear algebra solvers as libraries built into CUDA that are able to solve LU factorization, Cholesky factorization, and all other types of dense linear algebra problems [42, 128]. Work has also been conducted on systems that use both the CPU and the GPU to perform linear algebra problems in parallel on both processing units in libraries such as MAGMA [168]. While most QR factorization algorithms are inherently sequential, the most time consuming operations of

the algorithm can be parallelized to be executed on GPUs.

To allow for a wider audience to use GPU solvers, a considerable amount of research has gone into autotuning, methods that are used to automatically tune solver performance on different architectures [174, 110, 6]. Autotuning approaches cannot provide a certificate of optimality to ensure that they identify the best possible parameters since an exhaustive search of all parameter options is prohibitively expensive. Some common tuning approaches require a programmer's insight into the solver to determine acceptable values, and then use empirical testing to identify better values. Other methods use heuristics that limit certain parameters to a reduced set of values, and then exhaustively enumerate all of the remaining choices.

Optimal tuning parameters are desirable as they can lead to significant performance benefits. For example, well chosen parameters have led to a 25% increase in performance compared to default parameter values in the case of matrix-matrix multiplication [110]. This paper investigates the potential of using DFO and SO solvers to tune GPU algorithms for LLSPs. The idea of using DFO algorithms to tune algorithms is not new. Audet and Orban [16] proposed a DFO approach to tune a trust-region method. By using DFO and SO solvers to determine optimal tuning parameters for solving LLSPs with GPUs, the primary contributions of this paper are as follows:

1. We provide a computational comparison of DFO and SO algorithms, thus adding to a recently emerging literature on comparisons of DFO algorithms that is still in its infancy [147] and helps increase our understanding of the capabilities of these solvers.
2. We show that it is possible to accelerate MAGMA's QR solver by a factor of 1.8 for large TS matrices compared to the default MAGMA algorithm.
3. We demonstrate that a specific collection of five DFO solvers is capable of finding optimal GPU parameters and that it succeeds in doing so while requiring an order of

magnitude fewer simulations than complete enumeration.

The remainder of this paper is organized as follows. In Section 2.2, we review QR factorization which we use to solve dense LLSPs, and we analyze state-of-the-art CPU, GPU, and hybrid QR algorithms. In Section 2.3, we evaluate four QR solvers, cuSolverDN, LAPACK, MAGMA, and PLASMA in a comparative study that aims to determine a high quality solver that may be amenable to performance improvements through parameter tuning. In Section 2.4, we describe different types of DFO and SO algorithms that were used in our experiments to determine optimal GPU tuning parameters. In Section 2.5, we use the DFO and SO algorithms to tune MAGMA and compare the performance of the tuned MAGMA library against default options. Finally, we provide conclusions in Section 2.6.

2.2 QR factorization

QR factorization decomposes A into the product of two matrices Q and R, where Q is an orthogonal matrix, and R is an upper triangular matrix. This decomposition technique can be applied to any square or rectangular matrix. QR algorithms based on the Gram-Schmidt method and the Givens rotation method are sequential in nature and are not amenable to parallel computing. On the other hand, QR algorithms based on approaches such as the Householder transformation method can achieve high levels of performance on multicore computers or GPUs [5].

2.2.1 Parallel Householder factorization methods

Utilizing Householder transformations, a few parallel QR algorithms have been developed to solve large problems. These methods decompose A into panels or tiles that can be operated on with matrix-matrix multiplication that can be sped up with parallel computing. Implemented in LAPACK, panel factorization was introduced to speed up QR factorization

by taking advantage of cache locality [9]. Panel factorization is a technique where A is divided into rectangular panels which have m rows and n_b columns, where $n_b < n$. The block size, n_b, is an adjustable value, which is used to manipulate the algorithm's granularity. Blocked methods are composed of two operations, panel factorizations, which are limited by matrix-vector multiplication, and matrix updates which are bound by the performance of matrix-matrix multiplication. Panel factorization involves the decomposition of a panel into the product of Q and R, and matrix updates involve multiplying the Q^T matrix from the panel factorization with trailing panels. With the WY representation [155], it is possible to delay and apply many updates simultaneously with matrix-matrix multiplication.

2.2.2 Tall and skinny QR and communication-avoiding QR

While panel methods work well with square matrices that are limited by their update steps, blocked methods perform an order of magnitude worse for TS matrices [10]. This decrease in performance can be attributed to a communication limitation that occurs when the solvers are bottlenecked by memory bandwidth. To increase the performance of solvers for TS matrices, the authors of [55] created tall and skinny QR (TSQR) which is an algorithm that minimizes the amount of memory access required to perform QR factorization. Algorithms for TSQR divide the matrix into square tiles, and perform multiple tile factorizations simultaneously. Tile factorization is similar to panel factorization, except that tiles are of size $n_b \times n_b$ and elimination operations also need to be used to zero out tiles that are below the diagonal. After a series of tile factorizations, the transformations computed by the corresponding tile factorization are applied to the trailing tiles in each row of tiles. TSQR algorithms require more operations than panel factorization algorithms because of the need to use parallel tile factorization and elimination operations to generate R, which are not required in panel QR algorithms. However, factorization and elimination operations

can be overlapped in TSQR to increase performance when run on a multicore CPU or GPU.

The TSQR algorithm is designed for TS matrices but is slower than blocked methods on square matrices. As an extension to handle wider problems, communication-avoiding QR uses concepts from TSQR with a tree update procedure [10]. These changes allow TSQR to be extended for use on square matrix problems as well as on tall and skinny problems. In communication-avoiding QR, the block size is a key variable that controls the algorithm's granularity and trades off how small the tiles are with how efficiently a core can factorize a single tile.

2.2.3 State-of-the-art QR solvers

Motivated by the need to solve large and dense problems, QR factorization research has considered GPU-only, CPU-only, and hybrid implementations. Two state-of-the-art CPU solvers are LAPACK and PLASMA. LAPACK was one of the first dense linear algebra libraries created in the early 90's to handle large linear algebra problems [9]. PLASMA was created later as a multicore version of LAPACK that was meant to increase the parallelism of LAPACK and allow the solution of large problems more efficiently using blocking and other techniques. MAGMA, a hybrid dense linear algebra library [40], is designed to accelerate dense linear algebra routines by assigning sequential tasks to the CPU and parallelizable tasks to the GPU [168]. Hybrid algorithms gain significant performance benefits by simultaneously running operations in parallel on the CPU and the GPU. In particular, MAGMA runs the `dgeqrf` kernel from LAPACK on the CPU at the same time as the `magma_dlarfb` kernel on the GPU [93]. The LAPACK `dgeqrf` kernel performs a panel QR factorization routine on a panel of A, while the `magma_dlarfb` kernel uses the computed Householder reflectors from the factored panel to update the trailing matrix panels on the GPU. The `magma_dlarfb` kernel involves a series of matrix-matrix multiplications, and a triangular

matrix multiplication operation. These matrix multiplication operations are performed on the GPU through the cuBLAS library. Overlapping `dgeqrf` and `dlarfb` calculations, `magma_dgeqrf3_gpu` updates one panel. Simultaneously, the next panel is factored on the CPU and the remaining panels are updated on the GPU. Panel factorization computations are sequential in nature, and are best handled by the CPU, while update operations that are filled with matrix-matrix multiplication operations are best performed in parallel by the GPU, allowing for an efficient distribution of work.

One disadvantage of hybrid computing is that it relies on expensive data transfers between the processing units. All data that is transferred between these units has to pass through the PCI express bus. Newer PCI express buses have a peak memory bandwidth of 32 GB/s, an order of magnitude lower than the peak memory bandwidth of newer GPUs. This difference in transfer speed results in CPU to GPU communication being expensive, limiting the performance of many hybrid algorithms. Thus, hybrid algorithms have to be developed carefully to limit the transfer between the CPU and the GPU.

For problems where computations are not bottlenecking performance, solving the entire problem on the GPU can be a more efficient strategy. Communication-avoiding QR is one example of a GPU-only algorithm that is designed to avoid the penalty of the PCI express bus [10]. NVIDIA has also designed its own GPU-only dense QR factorization algorithm, included in the library cuSolverDN, where the compiler is in charge of optimizing architecture specific variables to solve QR factorization problems on the GPU [128]. One advantage of cuSolverDN is that NVIDIA has knowledge of how their software and hardware interact, allowing them to optimize their solver.

2.3 QR factorization comparative study

The standard approach for comparing QR factorization algorithms in the literature is to evaluate their performance on a range of square matrix problems [1]. Square matrices are used for comparing the peak performance of solvers when they are compute-bound, where solvers that have the best performance are considered the best. However, not all problems are able to be parallelized in an efficient way that can fully utilize the entire GPU. Thus, another meaningful performance metric is how well an algorithm performs when it is communication-bound (limited by data transfer). Communication limitations have been commonly observed when decomposing TS matrices with QR factorization [10]. While we compared different solvers for both square and TS matrices to learn which solvers are best for different types of problems, our primary focus was on discovering which solver was best able to handle TS problems.

We conducted our experiments on a workstation running CentOS7, on two Intel Xeon processors E5-2660 v3 at 2.6 GHz and 128 GB of RAM. The workstation is equipped with a NVIDIA Tesla K40 GPU, which has 15 streaming multiprocessors each with 192 CUDA cores, 12 GB of RAM, and a peak memory bandwidth of 288 GB/s. The algorithms are compiled with GCC version 5.2 using optimization flag -O3, and the NVCC CUDA 7.5 compiler when applicable. The matrices used in all of the experiments were randomly generated with elements between 0 and 1 from a uniform distribution. These matrices are sufficient for our purposes since performance of the algorithms compared is based entirely on the number of floating point operations performed, which is determined by the size of the matrix, and not its condition. Each matrix size was evaluated with ten different randomly generated matrices in double precision accuracy, and the average performances are reported. Performance was measured in terms of billions of floating-point operations per second (GFLOPs) taken by QR factorization. The number of operations needed to

solve LLSPs by $x = R^{-1}Q^Tb$ is negligible and ignored. We conducted two comparative studies, one on square matrices, and the other on TS matrices. Table 2.1 summarizes the solvers used in the computational experiments, and their defining properties. BLAS refers to basic linear algebra subprograms utilized and are the routines that perform basic vector and matrix operations [29].

Table 2.1: QR libraries used in comparative experiments

Solver	Computing environment	BLAS library
cuSolverDN v7.5 [128]	GPU only	cuBLAS v7.5 [42]
LAPACK v3.6.1 [9]	CPU only	OpenBLAS v0.2.18 [178]
MAGMA v2.1 [168]	Hybrid	cuBLAS v7.5
PLASMA v2.8.0 [78]	Multicore CPU	Intel MKL v16.0.3

2.3.1 Square matrices

We factor square matrices ranging from 1000 to 36000 rows by increments of 1000. Figure 2.1 shows solver performance in terms of GFLOPs as a function of matrix size. As seen in this figure, MAGMA outperforms all of the other solvers by over 150% for problems with more than 5000 rows. MAGMA, which simultaneously performs sequential factorizations on the CPU and update calculations on the GPU, excels at solving square problems that are amenable to solution approaches that utilize both processing units. Even though the performance of hybrid algorithms suffers from continuously transferring data between the CPU and the GPU, by overlapping updates with factorizations, MAGMA is able to minimize the idle time of the GPU allowing it to outperform the other solvers. While MAGMA is able to saturate the GPU with parallel update computations, cuSolverDN performs both update

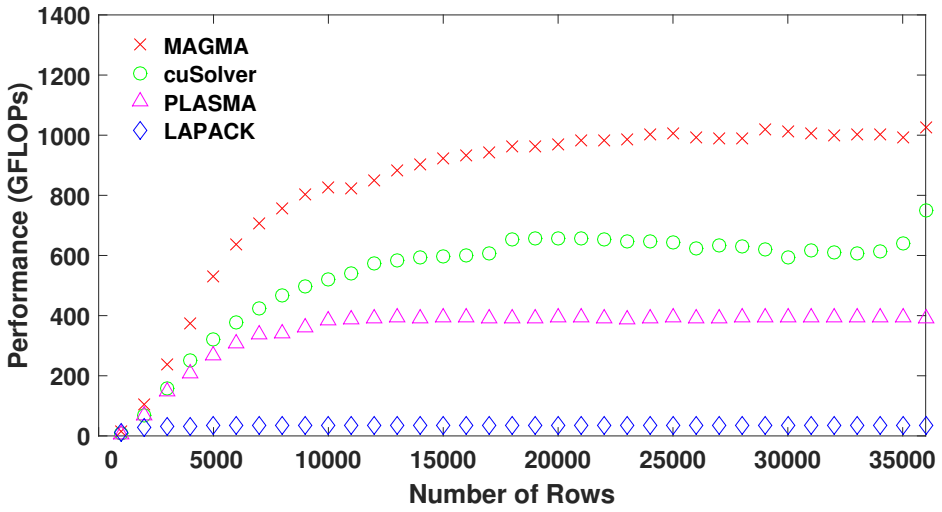


Figure 2.1: Performance of four different LLSP solvers for square matrix problems

and factorization operations on the GPU. When performed on the GPU, the factorization operations lower the average performance of the algorithm, and prevent devoting the entire GPU to the update operations that are rich in matrix-matrix multiplications. Both of these GPU solvers exhibit a typical behavior where their performance increases linearly as the problem size increases, until a certain point. Problems larger than that problem size begin to lead to leveling off in performance, because the GPU cores become fully utilized and peak performance is reached.

When comparing MAGMA or cuSolverDN to LAPACK or PLASMA, MAGMA and cuSolverDN are about ten times more efficient than LAPACK, highlighting the benefits of using GPUs for dense linear algebra. For larger square problems, PLASMA is five times faster than LAPACK, but PLASMA is still significantly outperformed by the GPU solvers because of how efficiently the GPU solvers are able to factor square dense problems.

2.3.2 Tall and skinny matrices

Even though the analysis of square matrices is useful for determining an algorithm's peak performance, TS problems are more relevant for solving LLSPs. Our choice of TS matrices also aims to analyze algorithms that are communication-bound because those problems are representative of LLSPs in the machine learning area [66, 44]. We are utilizing TS matrices with 500 columns and 1000 to 126000 rows in increments of 5000. Beyond 1000 columns, solvers become limited more by computations as opposed to communication limitations, which is not the behavior we are trying to investigate.

From Figure 2.2, the average performance of solvers on TS problems is an order of magnitude worse than the average performance on square problems with the same number of rows. This decrease in performance is related to solvers becoming communication-bound for TS matrices and is consistent with what has been observed before in the literature [10]. Algorithms are not able to perform as well on TS matrices because common approaches to solve these problems are bandwidth-bound and cannot fully utilize the GPU. Even though the performance for solving these problems is lower than for square problems, determining the best solvers for these problems is essential to efficiently solving LLSPs.

Based on the results in Figure 2.2, there is no clear best solver like in the case of square problems. For smaller problems, it is beneficial to solve the entire problem on the GPU as opposed to offloading small factorization operations to the CPU, which is why cuSolverDN has the best performance for these problems. Although cuSolverDN barely outperforms MAGMA, both GPU solvers outperform the CPU solvers. Like in the case of square problems, as the problems increase in size, the performance of the solvers begins to level off when the transfer rates become saturated. For problems with 35000 to 60000 rows, the performance of MAGMA increases and surpasses the constant performance of cuSolverDN, and remains above PLASMA.

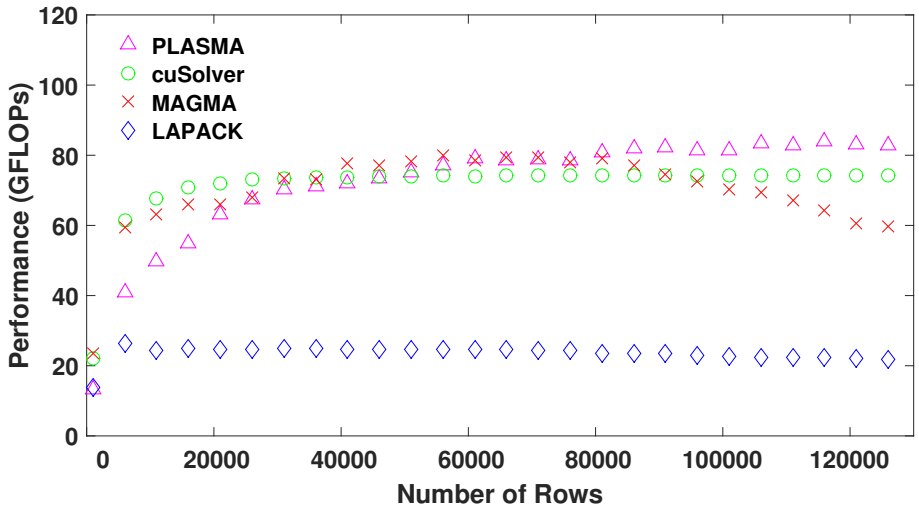


Figure 2.2: Performance of four different LLSP solvers for TS matrix problems

The most surprising result was that while MAGMA began to decrease in performance as the problem size increased beyond 60000 rows, PLASMA's performance increased. MAGMA's performance was expected to increase and level off as in the case of square problems. However, in this experiment as the number of rows increased, the number of columns was held constant. The block size which has a large effect on performance in MAGMA is controlled by the smallest dimension of the matrix. As the number of rows increased beyond a certain point, while maintaining the number of columns, the block size could not change leading to a decrease in performance for larger problems. Without changing the block size as the problems become even taller, the CPU became less efficient at decomposing taller matrices. This decrease in factorization efficiency, shifted the bottleneck from the update operations and data transfers on the GPU, that are normally the most time consuming, to the factorizations and data transfers on the CPU. PLASMA, on the other hand, experienced an increase in performance because it was able to parallelize both the factorization and update operations on the CPU without having to transfer data through the PCI express bus. The

performance of cuSolverDN does not seem to be affected by increasing the problem size, suggesting that the performance is probably limited by communication as opposed to how fast operations can be performed on the GPU. Finally, LAPACK had the worst performance of the four solvers for all problem sizes. Its performance seems to remain relatively constant regardless of the problem size.

These results suggest that, depending on the application, different solvers could be useful for solving TS LLSPs. One question we asked was if there was a way to increase the performance of MAGMA because it was able to achieve the highest performance in the square problems, suggesting that it could be the most efficient solver for large TS problems. In this study, we do not intend to tune all four of the solvers. Instead, we focused on only the one solver that seemed the most promising, MAGMA. Towards improving MAGMA, we investigated whether there was a way to not only stop the performance degradation on larger problems, but also to further increase the performance for all problem sizes above the other solvers. To answer this question, we carried out experimentations aiming to determine whether DFO and SO algorithms are capable of determining optimal tuning parameters for MAGMA. These optimization algorithms are discussed in the next section.

2.4 Derivative-free optimization and simulation optimization

As the complexity of GPU algorithms increases, the interactions between algorithms, compilers, and hardware becomes more difficult to model. As a result, the problem of tuning algorithms to particular software and hardware combinations has become challenging. Tuning algorithms is especially difficult when there is no explicit algebraic model for the relationship between tuning parameters and performance, forcing the one carrying out the optimization to treat the system as a black box.

To address these issues with a more systematic approach than has previously been used,

we utilized DFO and SO algorithms [147, 8]. These classes of algorithms address optimization problems whose objective functions are not available in algebraic form and/or gradients and functions are difficult, too expensive to evaluate, or noisy. Noise, for instance, may be due to random variations in measurements, including computer time measurements. Both classes of algorithms apply to black-box systems, i.e., systems for which the input-output relationship is not available in a form other than by querying a simulator or running an experiment. The literature reserves the term SO to denote algorithms that are built for an explicit treatment of noise and stochasticity. On the other hand, DFO algorithms may be applied to noisy problems but do not come with theoretical results for this class of problems. In the following sections, we will introduce the DFO and SO solvers we investigated in this study. For a more detailed background on DFO solvers, the interested reader is referred to [147], and to [8] for a more thorough analysis of SO solvers. Table 2.2 provides a listing of the DFO and SO solvers investigated in this paper.

Table 2.2: DFO and SO solvers used in this paper

Solver	Type
ASA† [94]	DFO, global, stochastic
BOBYQA† [143]	DFO, local, model-based
CMA-ES† [81]	DFO/SO, global, stochastic
DAKOTA/DIRECT† [3]	DFO, global, deterministic
DAKOTA/EA† [3]	DFO, global, stochastic
DAKOTA/PATTERN† [3]	DFO, local, direct
DAKOTA/SOLIS-WETS† [3]	DFO, global, stochastic
DFO† [41]	DFO, local, model-based
FMINSEARCH† [106]	DFO, local, direct

GLOBAL† [45]	DFO, global, stochastic
HOPSPACK† [137]	DFO, local, direct
IMFIL† [99]	DFO, local, model-based
MCS† [127]	DFO, global, deterministic
NEWUOA† [142]	DFO, local, model-based
NOMAD† [2]	DFO, local, direct
PRAXIS† [31]	DFO, local, direct
PSWARM† [172]	DFO/SO, global, stochastic
SID-PSM† [48]	DFO, local, direct
SNOBFIT† [92]	DFO/SO, global, deterministic
TOMLAB/CGO† [87]	DFO, global, deterministic
TOMLAB/GLB† [97]	DFO, global, deterministic
TOMLAB/GLC† [97]	DFO, global, deterministic
TOMLAB/GLCCLUSTER† [97]	DFO, global, deterministic
TOMLAB/LGO† [136]	DFO, global, stochastic
TOMLAB/MSNLP† [87]	DFO, global, hybrid
TOMLAB/RBF† [140]	DFO, global, deterministic
TOMLAB/GLCDIRECT* [87]	DFO, global, deterministic
TOMLAB/MIDACO* [87]	DFO, global, stochastic
TOMLAB/GLCFAST* [87]	DFO, global, deterministic
TOMLAB/GLCSOLVE* [87]	DFO, global, deterministic
TOMLAB/GLCCLUSTER* [87]	DFO, global, deterministic
SPSA BASIC† [160]	SO, local, stochastic approximation
SPSA Second Order† [160]	SO, local, stochastic approximation
STRONG† [37]	SO, local, response surface, trust region

SNM* [36]	SO, global, direct search

† solver accepts continuous variables only

* solver accepts continuous and integer variables

2.4.1 DFO local search methods

2.4.1.1 Direct methods

Direct search methods are defined as those that compare trial solutions with the current best solution using some strategy to determine what the next evaluation point should be [88]. In practice there is a variety of techniques that can be used, such as simplex methods, pattern search algorithms, and mesh adaptive direct search methods [125, 170, 15]. Originally, direct methods were used to solve difficult problems without any formal termination or convergence proofs [147]. As the field has developed, under certain assumptions, a few different direct methods can be shown to converge to a stationary point [169].

2.4.1.2 Model-based methods

Model-based methods sample the search space to generate surrogate models which can be used to suggest new evaluation points [147]. Surrogate models are typically first- or second-order models of the black-box system that can be solved to optimality faster than performing function evaluations on the black-box system. Surrogate models allow these methods to use gradient information from the surrogate model as well as probability density function information to guide the search of the true objective function. Model-based methods begin by sampling the search space to build a surrogate model. The surrogate models are then updated based on results from more evaluations of the black-box function. There are two major types of model-based methods, trust-region methods and filtering methods [71, 141].

2.4.2 DFO global search methods

2.4.2.1 Deterministic methods

Global deterministic search methods fall into two major categories, direct methods and model-based methods. These methods use techniques that balance local and global search so that they do not get trapped in a local optimum until they have sufficiently explored the search space. Some of these methods rely on optimizing a function that underestimates the objective function by using knowledge of the Lipschitz constant [157]. If the Lipschitz constant is unknown, then other methods to perform global search can be used such as DIviding the search space into hyperRECTangles (DIRECT), or branch-and-bound. The DIRECT algorithm computes function values at the center of intervals and searches the space for an optimal point, and in the absence of a Lipschitz constant terminates after reaching an iteration limit [97]. Multilevel coordinate search (MCS), like the DIRECT algorithm, divides the search space into boxes where it is able to vary how local the search method is by limiting how many times the same box can be subdivided and processed [91]. Branch-and-bound explores the problem space by branching on the domain. A lower bound can be estimated based on function evaluations and an upper bound can be determined through statistical bounds [135]. Then branches can be fathomed if a lower bound is no smaller than the best known solution.

Global model-based methods operate by optimizing a surrogate model to determine candidate optimal points for the black-box model. After a point is chosen from the solution of the surrogate model, that point is evaluated and used to update the accuracy of the surrogate. There are many techniques that can create these surrogate models such as response surface methods [19], pattern search [30], and optimization by branch-and-fit [92].

2.4.2.2 DFO stochastic methods

As opposed to deterministic methods, stochastic approaches require non-deterministic steps that can be used to choose evaluation points. Many stochastic DFO solvers are inspired by physical or biological principles. Stochastic methods have been widely studied in the literature and are simpler to implement than deterministic algorithms. Stochastic methods prevent getting trapped in a local optimum by using techniques to help them diversify their search strategy, while also being able to intensify and reach a global optimum when certain conditions are satisfied. A few of the stochastic search strategies have global convergence proofs under certain assumptions [20]. Some popular methods include hit-and-run algorithms [158], simulated annealing [121], genetic algorithms [86], and particle swarm optimization [57]. Many global derivative-free optimization solvers combine deterministic and stochastic methods into hybrid algorithms.

2.4.3 SO local search methods

Local SO algorithms rely on strategies that are similar to DFO, but include techniques that allow them to handle the uncertainty in the output values [8]. Three common types of these algorithms include response surface methodologies (RSM), gradient-based methods, and direct search methods. RSM relies on generating a surface (surrogate model) to model the input-output relationship of the simulation. Once a surrogate model is generated, derivative-based optimization techniques can be used to determine optimal points that can be compared with simulation results to further refine the model.

Finite difference is commonly used to estimate gradient information. However, in the case of simulation optimization, where simulations have uncertainty, and can be costly, finite differences is not an approach that can be used to identify accurate gradient information. Instead, gradient-based methods use stochastic approximations to generate an estimated

gradient to move towards an optimal solution. Simultaneous perturbation stochastic approximation (SPSA) is an algorithm that is able to estimate gradient information with only two function evaluations [160].

Direct search methods optimize by performing a sequential examination of points generated by some strategy [88]. These methods rely solely on a comparison of function values, and make no attempt to estimate gradient information. Almost all SO direct search methods are extensions of DFO direct methods, that employ sampling techniques to manage the uncertainty obtained in measurements, such as evaluating the same point a few times to calculate an average and standard deviation in the measurement. These methods are simple to implement and one example of an algorithm is SNM that uses a Nelder-Mead direct search [36].

2.4.4 SO global search methods

Global SO methods are comprised of ranking and selection algorithms, metaheuristics, model-based methods, and Lipschitzian optimization. Like DFO global methods, these algorithms are equipped with tools to escape from local optima and explore the entire search space at the cost of usually more function evaluations. With a finite parameter space, ranking and selection tries to minimize the number of repeated simulations required to accurately identify an optimal solution. The goal of these algorithms is to guarantee that the optimal design is better than all others by some amount δ with a probability of $1 - \alpha$. Metaheuristic approaches rely on some stochastic element when identifying what points to evaluate. Typical approaches used are genetic algorithms, simulated annealing, tabu search, and scatter search. These methods are easy to implement and often effective.

Model-based SO methods build probability distributions that are used to guide the search. For instance, covariance matrix adaption-evolution strategy (CMA-ES) generates proba-

bility distributions for covariances between variables, essentially amounting to estimating hessians. For a more exhaustive list of different strategies used for all of these methods see [8].

2.5 Using DFO and SO to optimize adjustable parameters of the MAGMA library

The aim of this computational study is to investigate if we can increase the performance of MAGMA by using DFO and SO solvers to tune its parameters. In this study, we investigated the effect of varying the block size (n_b) used in QR factorization and how A is stored in memory. The matrix A can either be stored in pinned memory on the CPU or in non-pinned memory allocated by malloc. By default, MAGMA stores A in pinned memory which can be faster to transfer data back and forth between the CPU and the GPU, but there is a cost associated with storing a matrix in pinned memory. Other typical GPU variables such as the number of threads, the size of a thread block, and the number of thread blocks to launch are primarily controlled by cuBLAS and were not able to be manipulated, unless we developed our own GPU BLAS routines. Typically, cuBLAS has proven to be the most efficient version of GPU BLAS and we defaulted to using that for this study.

Optimal tuning parameters are dependent on two factors, the GPU architecture that is being used, and the size of the matrix that needs to be solved. To determine optimal tuning parameters for any sized matrix, one can develop a lookup table where the matrix size and the GPU architecture are matched to optimal parameters. The MAGMA library has one of these lookup tables, which was created through experimentation on a different GPU architecture than the NVIDIA Tesla K40. To measure the effectiveness of using DFO or SO to determine tuning parameters, we compare the performance of using MAGMA default

parameters against the performance of parameters identified from DFO or SO. For the problem under study, there are 1000 possible combinations of parameters, thus affording us the possibility to determine an optimal solution via complete enumeration. The questions addressed in the computational experimentation were (a) whether DFO/SO solvers are able to find an optimal solution and (b) whether they can do so much faster than exhaustive enumeration.

2.5.1 DFO parameter tuning results

The DFO algorithms were used to determine optimal parameters for twenty-six different TS matrix sizes that were tested in Section 2.3. Each of the DFO solvers was given a limit of twenty function evaluations to search for an optimal solution. Most of the DFO solvers used in this experiment optimize in real spaces but the selected MAGMA parameters are integer-valued. Integrality was handled by rounding real values to the nearest integer inside the simulator. Each of the DFO solvers that uses a starting point was given the same initial point, and we carried out five trials with different starting points to minimize the effect a starting point had on solver performance. Parameters were selected from the DFO algorithm that produced the best result for each problem size. Trials were conducted with DFO, and MAGMA default parameters in the same manner as the previous experiments for TS matrices. To determine how far from the true optimal solutions the DFO solvers were, we also enumerated all feasible variable combinations. In Figure 2.3, it may appear counter intuitive that a few of the best DFO performance points are better than the enumeration performance. In these cases, the parameters reported for both the DFO and enumeration results were the same but differences in how the operating system prioritized CPU jobs resulted in slightly different timing measurements.

From Figure 2.3, we see that for smaller problems there is not much benefit in optimizing

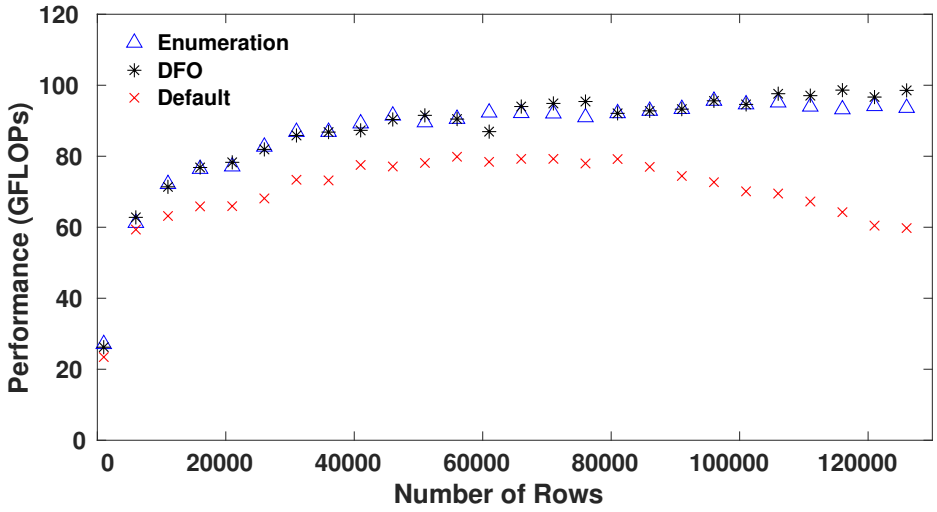


Figure 2.3: MAGMA's QR factorization performance with different tuning parameters

tuning parameters. As the problem size increases, we see significant performance benefits from finding optimal tuning parameters. For problems with 126000 rows, using the DFO determined parameters speeds up MAGMA by a factor of 1.8 in comparison to the MAGMA default values. Additionally, by using these tuned parameters, the performance of MAGMA not only does not decrease as the problem size increases, but instead continues to increase.

As seen in Table 2.3, as the problem size increased, optimal performance was found by decreasing the block size and using pinned memory. Using a smaller block size allows the GPU solver to more finely divide the work, and fully utilize the GPU for smaller problems. For all problem sizes, there is a performance increase compared to the MAGMA default values, but for large problems when the performance of MAGMA began to decrease, there is a significant performance benefit from using DFO-determined parameters. For every problem in our test set, the DFO solvers determined that using pinned memory always outperformed non-pinned memory. For smaller problems than those reported here we observed that using pinned memory is not optimal. Nonetheless, it appears that for

matrices larger than 700 by 500 pinned memory is the optimal way to store A.

Table 2.3: DFO determined optimal tuning parameters for different sized matrices compared against the default MAGMA values

Number of rows	Optimal n_b	MAGMA n_b
1000	76	64
6000	16	64
11000	16	64
16000	15	64
21000	13	64
26000	8	64
31000	8	64
36000	14	64
41000	8	64
46000	8	64
51000	8	64
56000	8	64
61000	8	64
66000	8	64
71000	8	64
76000	8	64
81000	8	64
86000	8	64
91000	8	64
96000	8	64

101000	8	64
106000	8	64
111000	8	64
116000	8	64
121000	8	64
126000	8	64

2.5.2 DFO solver performance

One initially puzzling result was that, for some problem sizes, the increase in performance from using DFO was not as large as for other problem sizes. When comparing the DFO results with the enumeration performance we observe that those performance decreases do not happen when the best parameters are chosen. This suggests that none of the DFO solvers were able to find the optimal parameters for certain problem sizes. However, this could be explained by the number of function evaluations given to the DFO solvers. Being able to determine high quality tuning parameters in a reasonable amount of time is a primary motivation for this work. To decrease the expected computation time, we spent some time experimenting with smaller problems. We first investigated what a sufficient number of function evaluations to determine good solutions was for a few problems. We determined that increasing the number of function evaluations above twenty did not lead to much of an observed performance benefit so we gave every DFO solver twenty function evaluations to determine optimal parameters. Although some solvers were able to identify good solutions in twenty function evaluations, some of the DFO solvers needed a few more function evaluations to determine optimal solutions and ended up with poor solutions after twenty function evaluations.

To determine what subset of solvers were best able to solve this problem, we compared

the performance of twenty-six continuous DFO solvers and five integer DFO solvers. If MAGMA would need to be tuned on another GPU, a user could focus on the solvers that proved to be the most efficient in this experiment, saving them time and allowing for high quality parameters to be determined quickly. Many of the solvers that we used in the experiment yielded parameters that had a worse performance than the MAGMA default performance. This could have occurred for a few reasons, the first being that we did not give enough function evaluations to some of the DFO solvers to determine optimal solutions. Without enough function evaluations, some of the solvers began evaluating points at one boundary that led to bad solutions, and were not able to find a good solution within the stated limit on function evaluations.

For local solvers in particular, another issue was that some of the solvers were getting stuck in fake local maxima. Using GFLOPs to measure the performance of MAGMA requires measurements of execution times. These execution times (wall clock times) can vary even when the same experiment is repeated multiple times. Noise in execution time could have resulted from the operating system prioritizing the algorithm slightly differently between runs, causing imbalances on how the problem was solved effecting performance. Noise effecting the DFO search was mostly observed for some local solvers, DFO, NEWUOA, FMINSEARCH, BOBYQA, and PRAXIS. These DFO solvers would begin at some starting point, and on the next function evaluation move to another point that would be rounded to the exact same initial point. The solver would then obtain a slightly different performance, and use the two different performance results from the same point to guide the search to a new point. For some of these solvers this process was repeated for all twenty function evaluations without the parameters being evaluated changing at all. The solvers would pick a value that was supposed to move towards a local minimum, but instead could not progress to an optimal solution.

In Table 2.4, the DFO solvers used were compared to demonstrate how well each performed for all of the TS problems. The columns in Table 2.4 indicate the average improvement that each solver had over the twenty-six problems tested, and the number of times each solver found the best block size. There were a few solvers that found the best parameters a few times, but performed fairly poorly on some problems decreasing their average improvement. Looking at both of these metrics, it is possible to identify a subset of solvers we can use to optimally solve all of the TS problems. If one were to use the five DFO solvers that had the best performance, GLCCLUSTER, HOPSPACK, SID-PSM, MCS, and SNOBFIT, they would be able to determine optimal tuning parameters for these twenty-six problems an order of magnitude faster than exhaustive enumeration could.

Table 2.4: Average improvement and number of best options that each DFO solver found in the TS matrix experiments

Solver	Performance improvement (%)‡	Best options
ASA	6	2
BOBYQA	-73	0
CMA-ES	-2	0
DAKOTA/DIRECT	7	0
DAKOTA/EA	4	0
DAKOTA/PATTERN	-16	0
DAKOTA/SOLIS-WETS	-11	0
DFO	-68	1
FMINSEARCH	7	0
GLOBAL	7	2

HOPSPACK	16	13
IMFIL	7	1
MCS	-2	2
NEWUOA	-73	0
NOMAD	4	1
PRAXIS	-74	0
PSWARM	-40	0
SID-PSM	13	10
SNOBFIT	3	4
TOMLAB/CGO	-16	0
TOMLAB/GLB	-16	0
TOMLAB/GLC	-16	0
TOMLAB/GLCCLUSTER†	-38	0
TOMLAB/LGO	-38	0
TOMLAB/MSNLP	-19	1
TOMLAB/RBF	-16	0
TOMLAB/GLCDIRECT	11	6
TOMLAB/MIDACO	-18	1
TOMLAB/GLCFAST	10	4
TOMLAB/GLCSOLVE	9	5
TOMLAB/GLCCLUSTER*	18	18

† solver accepts continuous variables only

* solver accepts continuous and integer variables

‡ Performance improvement calculated as $(t_{\text{default}} - t_{\text{tuned}})/t_{\text{default}}$

where t_{default} is MAGMA with default tuning parameters

Table 2.4 suggests that the continuous solvers that were best able to determine the optimal tuning parameters were local direct solvers SID-PSM and HOPSPACK, and global deterministic solvers SNOBFIT and MCS. We also observed that some of the integer solvers performed exceptionally well for these problems. For these problems, the integer version of GLCCLUSTER was able to find the optimal solution more often than any of the other DFO solvers, even for problems where none of the other DFO solvers could find the optimal solution. The other integer solvers, GLCDIRECT, GLCFAST, and GLCSOLVE all performed well and were able to identify the optimal parameters in a few problems, but were not able to perform as well as GLCCLUSTER. One surprising result is the comparison between the continuous and the integer versions of GLCCLUSTER. We notice an almost 60% increase in performance when the integer version is used, suggesting that there are significant benefits for using integer solvers in this parameter tuning problem. The integer solvers were able to do better than most of the continuous solvers because they did not get stuck on evaluating the same point because of rounding as in the case of continuous solvers.

One reason why global solvers are useful for solving these problems is that, if the functions are non smooth and possess many local maxima, local solvers may not be able to identify optimal solutions. In the case of our problem, when we carried out the enumeration of all the variable combinations, we discovered that the problems we are maximizing over are not well-behaved concave functions, as shown in Figure 2.4. Figure 2.4 was obtained by plotting the performance results from each parameter combination tested for a given problem size. The performance function has multiple local maxima. As a result, it would be difficult to find an optimal solution with local solvers. We believe the reason for the presence of multiple local maxima has to do with performance trade offs that occur when the block size is changed. In particular, through profiling of the MAGMA algorithm with different block sizes, we observed that, for large block sizes, the occupancy on the GPU was low, causing

most of the GPU to remain idle during a large portion of the calculations, while the data transfer rate was high. When the block size was smaller, we observed the opposite trend where the GPU had a high level of occupancy, and the transfer rate of data between the processing units was low. The places where local maxima occur are pareto optimal points that best maximized performance by balancing the trade off between the occupancy and data transfer rates between the CPU and the GPU.

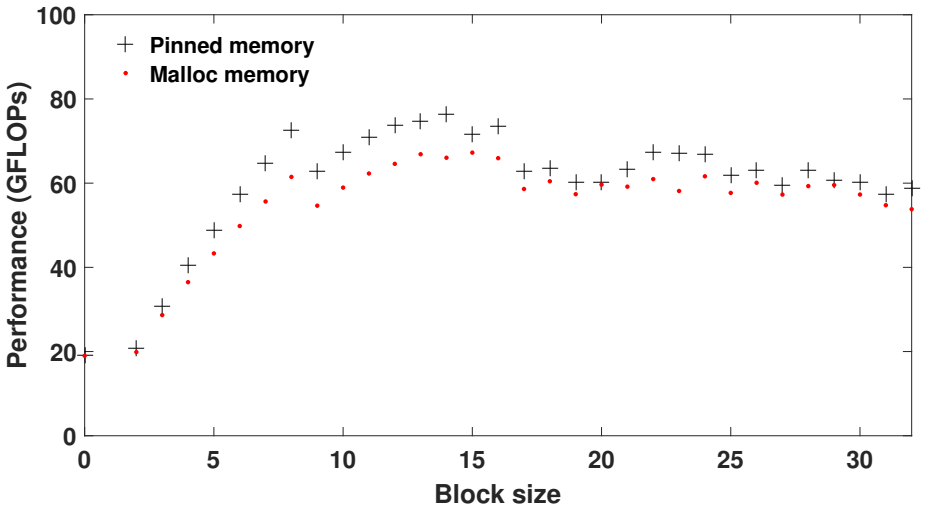


Figure 2.4: QR factorization performance for one problem with different options selected

In addition to tuning the parameters of each individual problem instance, we can search for better parameters to be applied to the entire test set. The best parameters from such a study would suggest a set of values that can be used as the default values in MAGMA. From this experiment, we discovered that by using a block size of eight and pinned memory, we were able to solve the entire test set faster than using the current default MAGMA parameters. Even though a block size of eight is not optimal for all problem instances, it leads to an average improved performance than using MAGMA's default block size of sixty-four. By using a block size of eight, users can improve their existing MAGMA QR solver

Table 2.5: Average improvement over MAGMA and number of best options that each SO solver determined with twenty function evaluations

Solver	Performance improvement (%)	Best options
SPSA basic	-74	0
SPSA 2nd order	-79	0
STRONG	-74	0
SNM	15	0

without having to use DFO to tune MAGMA. Of course, these values are only guaranteed to be optimal for the NVIDIA Tesla K40. Different values may be better for other GPU architectures.

2.5.3 SO parameter tuning

We used four different SO solvers: SPSA basic, SPSA 2nd order, STRONG, and SNM. In our first experiment, we ran the SO solvers under the same conditions as the DFO runs, i.e., using twenty function evaluations, and each solver was given the same initial starting point if the solver accepted one.

The average improvement over default MAGMA values of each SO solver is shown in Table 2.5. SO solvers not improving the performance of MAGMA was unexpected. SO solvers are designed for optimizing problems that have stochastic elements. However, three of the four SO solvers performed significantly worse than the default MAGMA parameter values. These results suggest that these three SO solvers require more function evaluations than DFO solvers to determine high quality solutions. On the other hand, SNM appeared to perform fairly well and found parameters that were improvements over the default MAGMA

values for each problem tested. However, for most problems SNM was only able to find suboptimal solutions.

Suspecting that SO algorithms typically require more function evaluations to determine optimal solutions, we also conducted an experiment where we allowed the SO algorithms 200 function evaluations. For the fourteen smallest problems we observed that, for every problem, more function evaluations had no effect on the performance of the SO algorithms. For the problems tested, the average performance benefit of using SNM increased from 8% to 12%, while the other solvers found solutions that on average caused MAGMA to take twice as long as MAGMA using default parameters. Both versions of SPSA, and STRONG found solutions that were heavily-dependent on the starting point they were given, while SNM was still only able to find suboptimal solutions. We conclude that these SO codes are not mature enough yet to handle problems of this complexity, causing them to not perform well, even with 200 function evaluations.

To compare the performance between SO and DFO solvers, we also studied the quality of the solutions found as the number of function evaluations increased. In Figure 2.5, we compared the best performance found by five different solvers over the first twenty function evaluations. These results demonstrate that the SO solver SNM was able to obtain a slightly better level of performance than the default MAGMA parameters within four function evaluations, and able to surpass that after sixteen evaluations. SO solvers typically will evaluate the same point multiple times to determine a mean and variance in the noise, explaining why the SO solver's performance remains relatively constant for twelve function evaluations, while the DFO solvers seem to be constantly improving performance.

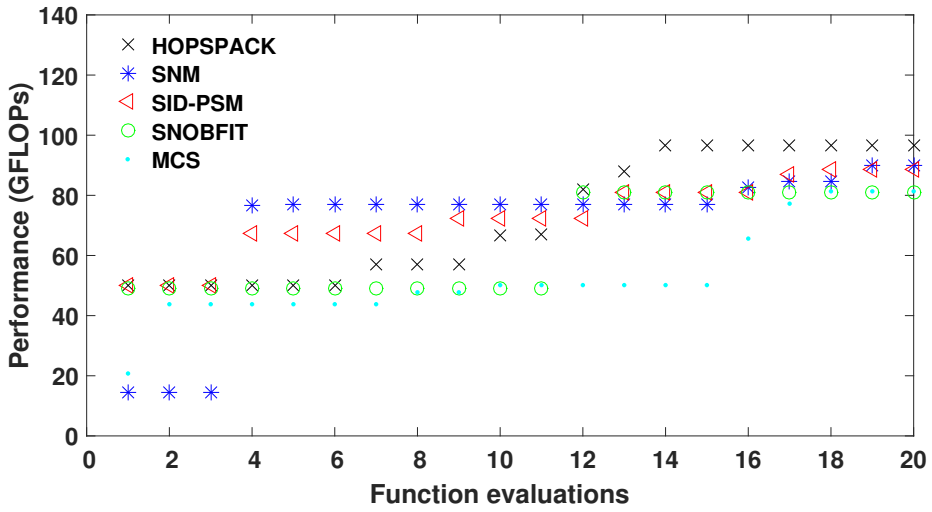


Figure 2.5: Tracking the best performance found as the number of function evaluations given to the different solvers increases for QR factorization of a 121000 by 500 matrix

2.6 Conclusions

This paper addresses the problem of determining optimal tuning parameters in GPU QR factorization. Previous attempts to perform tuning relied on heuristics combined with exhaustive enumeration to determine the parameters that maximized performance. We introduced a systematic approach based on using derivative-free optimization and simulation optimization algorithms.

The performance of thirty-one different DFO and four SO solvers was compared to determine solvers capable of optimizing the performance of the MAGMA library on a collection of problems. The best of these solvers provided optimal block size values that sped up MAGMA by a factor of 1.8 for tall and skinny matrices with over 100000 rows. With the DFO-determined parameters, the performance of MAGMA can be improved to above all of the other state-of-the-art solvers for TS matrix problems. Even for larger problems

that the untuned version of MAGMA with default parameters had worse performance than PLASMA and cuSolver, when tuned with DFO, MAGMA was able to outperform all of the other solvers. As seen in Figure 2.6, this parameter tuning made MAGMA the best performing LLSP solver over the entire range of tall and skinny matrices that we considered.

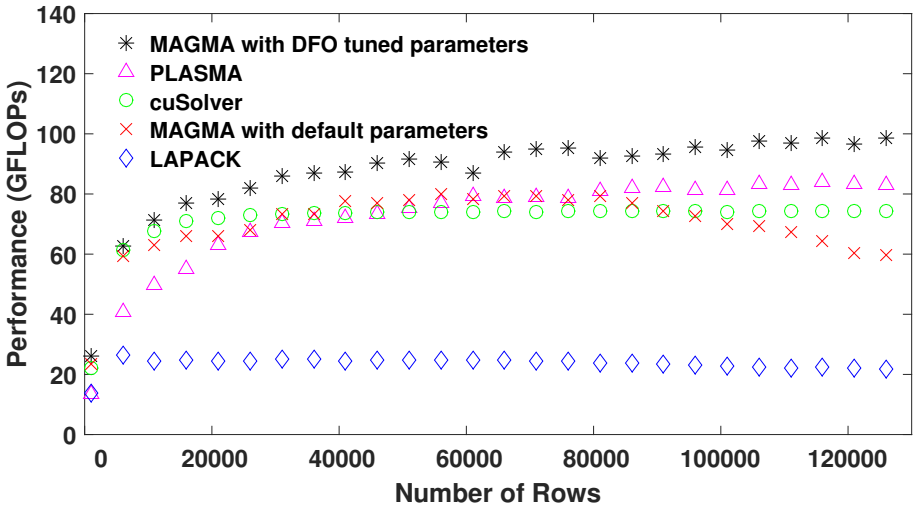


Figure 2.6: Performance of four QR solvers, compared against the performance of MAGMA with DFO-determined parameters

One of the reasons that we were able to find significant performance benefits over the MAGMA default parameters is due to the coarse grained nature of MAGMA's default lookup table. MAGMA only changes the block size based on the smallest dimension of the matrix, and the optimal parameters span a wide range of problem sizes. By improving this type of lookup table to include changes for every 1000 rows or columns, performance could be greatly improved. Even though this type of lookup table may not select the optimal parameters, those values would result in a better performance than the current default parameters and could also be used as an intial point for DFO or SO optimization to determine optimal tuning parameters for any problem size. This type of fine grained

lookup table will also allow a user the ability to try adjusting block size as factorization progresses without having to optimize every iteration to determine what the optimal block size may be.

The proposed approach relies on DFO solvers that are entirely agnostic to the hardware configuration used. As a result, single-GPU, multiple-GPU, as well as hybrid multi-core and multi-GPU environments can be tuned.

Chapter 3

HybridTuner

3.1 Introduction

As the landscape of high performance computing evolves, the need to design software that is optimal on a variety of computer architectures has grown. To meet this need, algorithms are designed with tunable parameters to allow for performance portability. Tunable parameters consist of categorical decisions, discrete options, or continuous values that impact algorithmic performance. Valid ranges for each parameter are determined from hardware specifications, developer insights, or desired algorithmic properties. Tuning is the problem of selecting a set of parameters that maximize the performance of an algorithm [163]. Identifying an optimal set of parameters in this space is challenging as it requires solving a multi-extremal optimization problem in the absence of an explicit algebraic function to relate input tuning parameters to an output performance metric.

To overcome the challenges in this problem, parameter tuning has been solved with derivative-free optimization techniques. Derivative-free optimization, also known as black-box optimization, is a field of non-linear optimization methods for addressing problems without an explicit algebraic function [147]. As solution strategies in this field involve querying a black-box, the number of simulations required to identify optimal solutions to a problem increases with the number of variables. In practice, tuning problems are challenging in cases with more than thirty variables, though some methods have been observed to find reasonable solutions for problems with hundreds of tuning variables [12]. An optimal set

of tuning parameters depends on the underlying computer architecture. We have observed that optimal parameters on one system do not provide any performance guarantees on another computer. In our computations, performance may decrease by up to 40% when parameters optimized for one architecture are employed on another. As there is no way to validate whether optimal parameters have been found for problems with a large search space, many users determine tuning parameters with heuristics, often producing inefficient algorithms.

Autotuning aims to automate the tuning process, where a programmer defines lower and upper bounds for all parameters and a search strategy selects the best set of parameters. Autotuners have been employed for compiler optimization, and machine learning applications. In machine learning, where hyperparameters affect classification accuracy, Bayesian Optimization is commonly utilized for hyperparameter tuning [17].

One area of research that has made use of autotuning is the deployment of algorithms with graphics processing units (GPUs). GPU tuning problems have a highly nonlinear and discontinuous parameter space that proves challenging for local search methods and heuristics to optimize [151]. As the landscape of GPU architectures is constantly evolving, selecting the best tuning parameters requires a thorough understanding of hardware and knowledge obtained from performance profilers. Additionally, specific GPU parameters such as block size, thread block size and grid size have been reported to have dramatic effects on performance. Well-tuned GPU parameters can reduce execution time by a factor of 1.2x to 1.4x [110, 151]. Further considerations have to be given to efficiently transferring data between the CPU and GPU and how discrete decisions impact performance. All of the factors above contribute to a large parameter space with discrete choices, requiring the use of autotuning techniques for efficient performance optimization.

A problem of particular interest is optimizing dense linear algebra kernels such as matrix

multiplication. As the most time consuming operation in several algorithms, the performance of matrix multiplication directly impacts the performance of algorithms such as training neural networks through backpropagation, solving graph theory problems and solving linear and nonlinear optimization problems. Therefore, matrix multiplication speedups are essential for scaling up to solve more complex problems. While the problem of parameterizing and tuning dense matrix-multiplication has been investigated before [110, 53, 123, 162], it has been addressed primarily with heuristic approaches. None of these heuristic approaches provide optimality guarantees, and offer limited insights except possibly on the architecture the algorithm is designed for. Tuning dense matrix-matrix multiplication is challenging, as the search space is nonsmooth, nonconvex, and too large to be solved with exhaustive enumeration.

Parameter tuning is also important in the area of compiler optimization. The GCC compiler has over three hundred optimization flags and parameters. The performance of algorithms compiled with GCC is affected by the choice of optimization flags or the parameters selected [14, 12, 164]. While default optimization flags -O2 or -O3 are commonly used, the authors of OpenTuner observed that with tuning they were able to reduce execution time by a factor of 2x compared to default optimization flags when tuning a matrix multiplication algorithm. There have been many proposed methods for tuning GCC, including the use of machine learning techniques [13].

In this work, we investigate the benefits of new autotuning strategies based on hybridizing derivative-free optimization algorithms. The primary contributions of this paper are as follows:

1. We propose an algorithmic framework to combine local and global DFO approaches for parameter tuning. Two methods, Bandit DFO and Hybrid DFO, are introduced that identify the best or near-optimal solutions for all problems that are considered

in this work.

2. We demonstrate that, by combining local and global DFO strategies, it is possible to improve the performance of dense matrix multiplication by a factor of 1.4x compared to optimal parameters identified by OpenTuner [12], ActiveHarmony [163] and Bayesian Optimization [17]. Bandit DFO also optimizes the performance of algorithms compiled by GCC, identifying the best observable parameters with fewer than 5% of the iterations required by other methods.
3. We share our implementation with the community facilitating the development and use of hybrid autotuning algorithms. We provide the proposed Bandit DFO and Hybrid DFO as free and open-source software available at https://github.com/bsauk/HybridTuner. The proposed methods can be used with any derivative-free optimization solvers. We include several open-source DFO solvers in our implementation.

In Section 3.2, we review related literature, including the field of autotuning, the literature on derivative-free optimization algorithms, and hybrid tuning algorithms closely related to this work. In Section 3.3, we propose hybrid DFO algorithms and describe advantages of hybrid methods over other approaches. In Section 3.4, we present a computational comparison between autotuners and our proposed hybrid methods to measure the performance of the proposed method for tuning matrix multiplication and the GCC compiler. We provide conclusions in Section 3.5.

3.2 Literature Review

3.2.1 Autotuners

Algorithmic parameter tuning or autotuning has been studied over the last two decades [163]. The goal of autotuning is to determine a set of algorithmic options that maximize the per-

formance of an algorithm for solving a given problem on a specific computer architecture. Algorithmic options may consist of loop unrolling, blocking, parallelization schemes, or a set of discrete solution strategies [11]. While small modifications to a code lead to significant performance improvements, in some cases, a code may need to be completely rewritten to further improve performance. For a majority of programmers who utilize code developed by others, determining values that seem to be defined arbitrarily is a daunting task. Autotuning attempts to simplify the process of developing efficient algorithms with a variety of techniques ranging from automating code generation to local direct derivative-free optimization search methods.

Many authors pose the tuning problem as a black-box optimization problem, that can be solved with local direct derivative-free optimization methods. These direct methods use an intelligent search strategy to select a point to evaluate, using data from previously evaluated iterations. ActiveHarmony is one of the first autotuners developed [163], which uses a Nelder-Mead simplex search strategy to suggest points to evaluate that may be potentially optimal. PetaBricks generates several algorithms from a list of algorithmic options, and automates code generation from provided inputs [11]. Algorithms tuned by the PetaBricks compiler are divided into sub-problems that are tuned independently of each other using a genetic algorithm. OpenTuner is an autotuner that combines several local search strategies to determine high quality solutions [12], by solving the multi-armed bandit with sliding window, area under the curve credit assignment problem (AUC Bandit Meta Technique) [133]. This technique balances exploiting strategies that have performed well in recent iterations with the potential benefit of having other strategies explore the search space. Other autotuning approaches hide the tuning process from users. These autotuners are designed for domain-specific languages where each autotuner is developed for one specific application, and optimizes an algorithm given assumptions that must hold true for the particular applica-

tion of interest. Automatically Tuned Linear Algebra Software (ATLAS) [177], is a software that was developed by Whaley et al. to automate the empirical tuning and optimization of basic linear algebra subroutines (BLAS). ATLAS optimizes the performance of BLAS on any computer, regardless of architecture, without requiring an extensive knowledge of the system or the underlying linear algebra involved. For BLAS and LAPACK libraries, algorithmic developers have developed autotuning approaches for their algorithms based on heuristics and experimental observations to allow for performance portability [110]. The Optimized Sparse Kernel Interface (OSKI) [176] improves the performance of sparse matrix kernels. OSKI uses heuristics to tune sparse algorithms automatically without any input.

In the machine learning literature, hyperparameter tuning refers to techniques to select parameters for neural networks, such as learning rate, momentum, and categorical decisions, such as the type of activation function to select. Some parameters have been observed to affect the accuracy of classification models produced by neural networks or support vector machines. As accuracy is affected by the selection of hyperparameters, considerable effort has been invested in hyperparameter optimization. Tuning these algorithms is challenging and is typically addressed with random search, grid search, or with Bayesian Optimization using Gaussian Process models [23]. Bayesian Optimization is widely regarded as the most efficient way to perform hyperparameter tuning [23, 159, 17]. Other stochastic strategies, such as the Covariance Matrix Adaption Evolution Strategy, have been observed to outperform Bayesian Optimization solvers as the iteration budget increases [113].

3.2.2 Derivative-free optimization algorithms

DFO algorithms are divided into groups depending on how they search for an optimal solution. One such distinction is between direct methods, such as the Nelder-Mead simplex [125], that search over a certain pattern and model-based methods, such as the trust

region method [141], that rely on a model to guide the search. DFO solvers are also classified by whether they search for local or global solutions, and whether they are deterministic or employ stochastic elements. For a more detailed discussion of DFO solvers, readers are referred to [147]. In this work, several DFO solvers that are investigated are listed in Table 3.1, and expanded upon in the remainder of this section.

Table 3.1: DFO solvers considered

Solver	Type
HOPSPACK [137]	local, direct
SID-PSM [48]	local, direct
SNOBFIT [92]	global, model
DAKOTA MESH ADAPTIVE SEARCH (MADS) [4]	local, direct
DAKOTA SOGA [4]	global, stochastic
TOMLAB/glcDirect [87]	global, deterministic
TOMLAB/glcFast [87]	global, deterministic
TOMLAB/glcSolve [87]	global, deterministic

HOPSPACK [137] is a direct solver that uses a generating set search (GSS) procedure [105]. GSS was proposed as a type of grid search algorithm for DFO. In particular, HOPSPACK implements an asynchronous version of GSS such that multiple trial points are evaluated simultaneously and, when a partial improvement is observed, new trial points are proposed around that solution. While this approach requires more function evaluations, promising points are identified faster than with a synchronous approach, where only the best trial points are used in subsequent iterations. HOPSPACK has the capability to use several search strategies. To facilitate the transfer of information from one strategy to an-

other, evaluated points are stored in cache memory to prevent repeating a trial point that has been evaluated by a different solver.

SID-PSM [48] is a MATLAB implementation of a pattern search method for solving constrained or unconstrained nonlinear optimization problems [47]. The authors improve upon the generic pattern search algorithm by incorporating a quadratic minimum Frobenius norm model into the direct search method [46]. SID-PSM has been proven to have global convergence to a stationary point when solving unconstrained nonlinear problems [170].

Stable Noisy Optimization by Branch and FIT (SNOBFIT) [92] is a global model-based algorithm that combines randomization with surrogate models to identify an optimal solution. The SNOBFIT package fits linear and quadratic surrogate models around sampled points. A quadratic model is fit around the incumbent solution, while linear models are fit around all other points. SNOBFIT determines the next point to sample by optimizing over surrogate models and then evaluating at the point that is expected to minimize the black-box function. Randomization is incorporated by sampling additional points from unexplored areas of the search space.

DAKOTA Mesh Adaptive Search (MADS) [4] is a DFO generalized pattern search. Under mild assumptions, MADS has been shown to converge to a stationary point when the set of evaluated points becomes dense. MADS can solve both unconstrained, bound constrained, and nonlinear constrained problems, and optimizes problems with continuous, discrete, and categorical variables. The mesh adaptive search is available in the NOMAD software [2].

DAKOTA Single-Objective Genetic Algorithm (SOGA) [4] is a stochastic global optimization DFO algorithm. Genetic algorithms optimize by generating a population of candidate solutions, evaluating the population, and advancing through generations of a population. In each generation, there can be crossover, causing two candidates to exchange information, mutations, where random variations take place, or the updated solution pool is evaluated.

A population is evaluated until the algorithm converges or the number of iterations reaches a predefined iteration limit.

TOMLAB glcDirect, glcFast and glcSolve [87] are all implementations of the DIviding RECTangles (DIRECT) algorithm [96]. DIRECT is initialized by sampling the center point of the search space. Then, a variable is selected and its range is divided into three segments of equal length. The parameters at the center of the three regions are evaluated. Then, depending on which region has the best performance, another variable is selected and trisecting continues in the most promising region. The search procedure is repeated, continuing to refine the sampled points until a budget on the number of function evaluations is exhausted.

3.2.3 Existing hybrid tuning algorithms

Hybrid algorithms combine simple solvers that may otherwise be unable to escape from a local optima. Hybrid search algorithms have been demonstrated to be effective at identifying optimal tuning parameters for different computational algorithms [12]. In several cases, local DFO solvers have been combined with global strategies to improve their accuracy with fewer function evaluations [63, 173].

One line of research involving deterministic hybrid tuning algorithms combines the DIRECT algorithm with implicit filtering [35], pattern search techniques [77], or with surrogate models [85]. In the work of Hemker et al. [85], in every iteration, instead of evaluating a point at the center of a box, as is done in DIRECT, the point to be evaluated is determined by minimizing a surrogate over the area of the domain that is being considered. The authors test this implementation on a problem with 17 variables and bound constraints, but found that the hybrid implementation was unable to determine an optimal solution after 400 function evaluations.

Griffin et al. [77] developed a hybrid framework that combines DIRECT and APPSPACK [76]. APPSPACK is a predecessor of HOPSPACK and is also a generating set search DFO solver. The work of Griffin et al. focuses on solving small problems with two, three, or four variables, and describes how the framework scales with an increasing number of CPU cores. The authors utilize the asynchronous nature of the DIRECT algorithm, and parallelize the search process. Their algorithm does not use DIRECT to initialize the starting point of the APPSPACK algorithm, but instead interleaves the execution of the two algorithms throughout the entire search.

OpenTuner is a hybrid strategy that is initialized with a random starting point, and then explores the search space with several local search strategies [12]. The multi-armed bandit problem is maximized after every iteration to select a solver to locate the next trial point. OpenTuner has been demonstrated to perform well for parameter tuning on high-dimensional problems, such as tuning GCC, while it identifies optimal solutions for smaller problems within a fraction of the iterations required by exhaustive search.

3.3 Proposed hybrid tuning algorithms

Our proposed methodology derives from the observation that certain global DFO strategies identify high quality solutions quickly, but then struggle to escape from a local minima to obtain a globally optimal solution. We propose two hybrid methodologies, Bandit DFO and Hybrid DFO. We create Bandit DFO by combining a global DFO solver with several local DFO solvers that are selected by solving the multi-armed bandit function. In Hybrid DFO, we initialize a local DFO strategy with the solution found by a global DFO strategy that is executed for a small fraction of the computational budget.

The methods we propose here borrow ideas from OpenTuner, but significantly improve tuning performance by combining local with global DFO solvers. Additionally, merely

implementing a global DFO solver in the OpenTuner framework would be insufficient to obtain near-optimal solutions as quickly as the proposed Bandit DFO method. A key improvement in our implementation is the frequency at which we solve the multi-armed bandit problem. We identified that solving the bandit problem less frequently results in faster convergence to optimal solutions, when including model-based DFO solvers.

3.3.1 Multi-armed bandit technique

The multi-armed bandit problem with a sliding window, area under the curve credit assignment problem is a technique developed for programmatic autotuning [133, 12]. This problem is inspired by attempting to maximize profit when gambling on numerous slot machines [64]. If multiple machines are available, initially, one has to sample each of the machines to discover the expected profit from each game. Then, there is a balance between exploiting machines that have worked well with exploring options that have not been evaluated recently.

This framework translates well into a hybrid autotuner. In hybrid algorithms, there is a trade-off between exploiting solvers that are currently performing well, and exploring the potential of other solvers with unknown or previously poor performance. Different techniques have been proposed in the literature to balance the trade-off between these objectives, such as simulated annealing [121] and particle swarm optimization [100].

The multi-armed bandit problem is solved at every iteration in the OpenTuner framework to select which local solver to use at the current iteration. This approach also includes a sliding window to bias the current selection towards solvers that have performed well recently, while ignoring results outside of the current time window. The following optimization problem is solved to determine which technique (t) to select:

$$\underset{t}{\operatorname{argmax}} \frac{2}{N_t(N_t+1)} \sum_{i=1}^{N_t} iV_{ti} + C\sqrt{\frac{2\log W}{H_t}}$$

Here, N_t is the number of times that technique t has been used in the current time window, and V_{ti} is 1 if the use of technique t in time period i resulted in discovering a better solution. The length of the sliding window is defined as W. H_t is the frequency of using technique t in the current time window. Finally, the non-negative parameter C controls the exploration and exploitation trade-off. Large values of this parameter put more emphasis on exploration; smaller values on exploitation. Traditionally, this maximization problem is solved at every iteration to determine the solver that is expected to perform the best in the next iteration.

While solving the bandit problem at every iteration may be appropriate when using local direct DFO methods, in practice, we observed that model-based DFO strategies require several iterations before suggesting a better candidate solution. To allow model-based DFO methods to find better trial points, we introduced another hyperparameter, n, into the algorithm to control the frequency of solving the multi-armed bandit problem. This modification is critical to the performance of our algorithm, and can be set to one to recover the original formulation. Figure 3.1 outlines our multi-armed bandit function implementation. During initialization, it is possible to select a starting point either manually, or by using a DFO solver for a small number of function evaluations. In this work, we initialize Bandit DFO with the DIRECT search method. The hyperparameters in this implementation are:

- C to control the exploration and exploitation trade-off;
- Max_{Evals} is the tuning function limit;
- W is the number of iterations to consider in the sliding window; and
- n is the number of iterations between solving the multi-armed bandit problem.

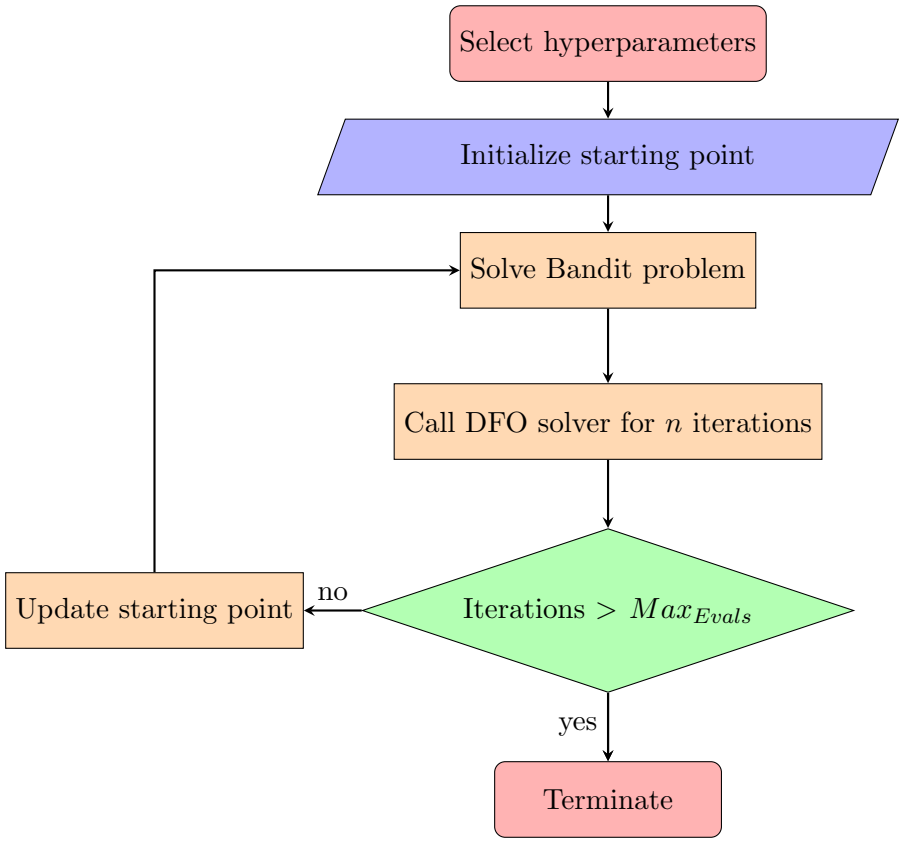


Figure 3.1: Algorithmic framework of proposed Bandit DFO method

3.3.2 Initialization strategy

The proposed Hybrid DFO algorithm is outlined in Figure 3.2. It has previously been observed that the DIRECT algorithm locates near-optimal solutions with a small number of iterations but is unable to converge to a globally optimal solution. We improve the performance of DIRECT by initializing a local DFO solver with the solution returned from DIRECT after a small number of iterations. DIRECT has been combined with local optimizers before, but its application was limited to small problems [96, 76]. To extend this idea to large-scale problems, we investigated what percentage of the experimental budget to allocate to global search to identify a near-optimal solution.

We initialize the HOPSPACK and SID-PSM algorithms with a starting point identified from TOMLAB glcDirect. While HOPSPACK and SID-PSM are local direct search DFO solvers, they have been observed experimentally to identify the best measured solutions in previous tuning experiments [151]. We have observed that initializing these solvers with an intermediate solution obtained by DIRECT, leads to faster convergence to near-optimal solutions in comparison to other similar techniques. From experimentation, we identified that assigning $5\% - 10\%$ of the iteration budget to a global DFO method leads to near-optimal performance with this hybrid approach. Allocating global solvers $5\% - 10\%$ of the computational budget corresponded to approximately two or three iterations for each tuning problem that was considered in this work. The hyperparameters that we consider are:

- n_{Global} controls the number of iterations given to DIRECT; and
- n_{Local} is the number of iterations given to a local DFO solver.

In the next section, several state-of-the-art autotuners are compared against our proposed hybrid approaches. In the first subsection, all algorithms are used to tune a customized GPU matrix-matrix multiplication algorithm with 17 tuning parameters. Then, we tune GCC compiler parameters and optimization flags on three different applications from the PolyBench benchmark suite [179].

3.4 Computational Results

We conducted computational experiments on two different machines. The first, running CentOS7, with an Intel Xeon E5-1630 at 3.7 GHz and 8 GB of RAM, with a NVIDIA Tesla K40 GPU with 15 streaming multiprocessors, 12 GB of RAM, and a peak memory bandwidth of 288 GB/s. Algorithms were compiled with the NVCC CUDA 9.1 compiler or the GCC 4.8.5 compiler when applicable. For the other experiments, the Pittsburgh

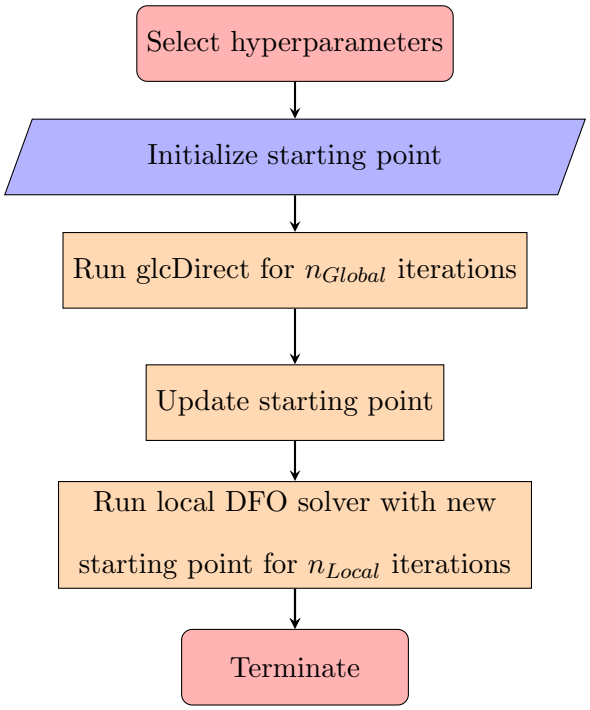


Figure 3.2: Algorithmic framework of proposed Hybrid DFO method

Supercomputing Center, Bridges, was used to perform experimentation on a NVIDIA Tesla P100 GPU [131, 171]. Dense matrix-matrix multiplication is performed on each GPU with a matrix multiplication algorithm developed by modifying example code provided by NVIDIA in the CUDA 9.1 release to allow for the inclusion of tunable parameters. We created the modified example code to create a tuning space that considers algorithmic options and NVCC compiler optimizations. The parameter space for this problem is $3.4x10^{11}$ unique combinations. The GCC examples are codes from the PolyBench 4.2.1 benchmark suite [179].

Given the size of the parameter space and that the objective function is a black-box, we are unable to enumerate all parameter combinations to determine an optimal set of tuning parameters for each problem. Instead, we compared solvers by their performance

relative to the best solution found in each experiment. Solvers that identify parameters that produce the highest observable performance in the shortest number of function evaluations are regarded as the best autotuner in each experiment. In all of the figures provided below, Bandit refers to our Bandit DFO algorithm, Hybrid refers to the Hybrid DFO algorithm, and Bayesian refers to solving the problem with Bayesian Optimization.

3.4.1 Matrix multiplication on the Tesla K40

In the matrix multiplication experiment, we addressed the problem of tuning 17 parameters. These parameters are listed in Table 3.2, along with their corresponding lower and upper bounds. Parameters consist of categorical choices and integer decisions. Categorical choices are represented as 0 or 1 binary decisions such as whether to use GPU shared memory. Integer choices include methods for optimizing spatial locality through tiling. GPU specific parameters, such as the number of threads in a thread block, are adjusted by varying an integer value, while maintaining hardware constraints. The inner loop of matrix multiplication is unrolled based on the loop unrolling parameter. The remainder of the parameters are NVCC compiler optimizations. We considered the parameters that the authors of [33] identified as NVCC parameters that could be tuned to outperform the -O2 or -O3 compiler flags. All parameters investigated in this experiment are integer variables. As several DFO solvers only operate on continuous variables, and may attempt to evaluate fractional trial points, we rounded values to the nearest integer before passing them to the matrix multiplication kernel.

We report results only for square matrices of size 10000 by 10000. Not shown here are results from additional experiments that we conducted with 2000 by 2000 and 6000 by 6000 matrices, for which we observed similar trends to those of the figures shown below on both the Tesla K40 and the Tesla P100 GPUs. The tuning objective was to maximize

Table 3.2: Algorithmic options for tuning dense matrix-matrix multiplication

Tunable parameter	Lower bound	Upper bound
Store transpose of matrix A	0	1
Store transpose of matrix B	0	1
Use of shared memory	0	1
Block size	1	63
Number of threads in x direction	1	32
Loop unrolling	1	256
No-align-double	0	1
Relocatable-device-code	0	1
Single-precision denormals support	0	1
Single-precision floating-point division	0	1
Single-precision floating-point square root	0	1
Cache modifier on global load	0	2
Optimization level	0	3
Fusion of multiplication and addition	0	1
Allow expensive optimizations	0	1
Maximum amount of register count	24	63
Preserve resolved relocations	0	1

performance of the matrix multiplication kernel. Performance was measured in gigaflops (GFLOPs) and was calculated as the number of operations required to perform dense matrix multiplication divided by the amount of time required to perform all of the operations. All solvers were allowed to call the matrix-matrix multiplication kernel 1000 times, and the

best performance obtained in the experiment was reported against the iteration number in the figures below. The same random starting point was given to all solvers, if an initial starting point was accepted.

As shown in Figure 3.3 for multiplication of 10000 by 10000 matrices, we observed that the autotuners have different performance profiles. ActiveHarmony terminated before finding a solution above 200 GFLOPs, less than half of the peak performance obtained by our Bandit DFO. OpenTuner identified a solution with a performance of 350 GFLOPs after 300 function evaluations, using 10 times more function evaluations than the Hybrid DFO solvers to obtain a similar result. The Bayesian strategy has a similar performance to Bandit DFO for the first 400 iterations. However, the algorithm was terminated by the operating system on our machine because it ran out of memory after 500 iterations.

The proposed hybrid initialization technique discovered a solution with a performance over 300 GFLOPs within the first 20 iterations. Hybrid DFO never escaped from the locally optimal solution initially obtained by the TOMLAB solvers and terminates after 400 iterations. Both Hybrid DFO and Bandit DFO use a DIRECT algorithm initially. However, Bandit DFO explored a different search direction than Hybrid DFO, leading to a worse performance for the first 600 iterations. After 550 function evaluations, the SID-PSM sover in Bandit DFO escaped from the previous local optima and improved to over 400 GFLOPs. Bandit DFO converged to a parameter set that outperformed the best solution obtained by the other autotuners by more than 80 GFLOPs after 600 iterations.

3.4.2 Matrix multiplication on the Tesla P100

We also performed the same set of matrix multiplication experiments on another GPU, the Tesla P100. Figure 3.4 displays the results for the multiplication of square 10000 by 10000 matrices. ActiveHarmony and OpenTuner obtained suboptimal solutions and ter-

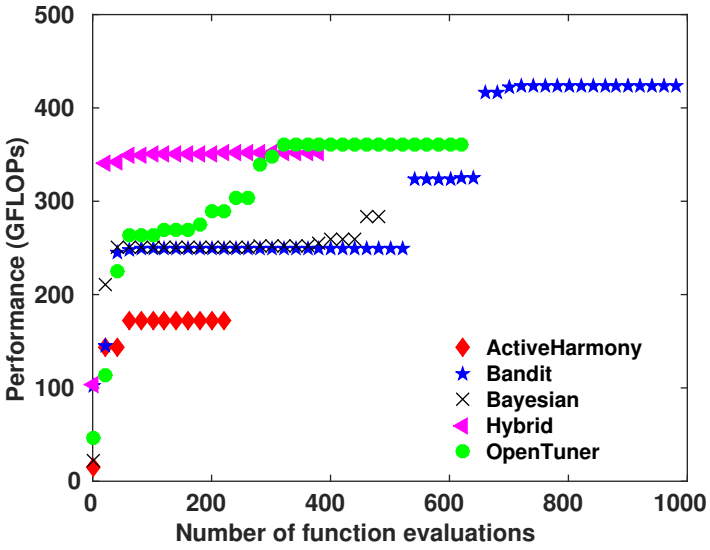


Figure 3.3: Comparison of the performance of different autotuners tuning dense matrix multiplication for 10000x10000 matrices on the Tesla K40 GPU

minated after 400 and 700 function evaluations respectively. Unlike on the Tesla K40, Bayesian Optimization was able to be run for 1000 iterations on the Pittsburgh Supercomputing Center Bridges. While we do not report all of the results here, we note that the Bayesian strategy had the same performance profile on all three of the matrix sizes that we experimented on, converging to a performance around 1600 GFLOPs. In each case, the Bayesian approach arrives at a slightly different set of optimal parameters, even though the performance was similar.

Bandit DFO was the first to achieve a performance over 2000 GFLOPs, and then outperformed all of the other solvers after 550 function evaluations. The use of multiple DFO solvers combined with the DIRECT strategy performed well in both obtaining near-optimal solutions, and converging to the best solution observed in our experiments. The second-best performing strategy was our Hybrid DFO algorithm. This algorithm was the fastest solver to find a solution with a performance over 2100 GFLOPs. Our main results on the P100

GPU align with the results on the K40 GPU. Parameters obtained by using our hybrid tuning algorithms are superior to those obtained with other autotuners, either in terms of performance, or in the number of function evaluations required to identify the best known solution. Near-optimal solutions are obtained within the first 200 iterations with the proposed algorithms, while OpenTuner, ActiveHarmony, and Bayesian Optimization fail to find near-optimal solutions after 1000 iterations. From experiments conducted here, our proposed hybrid methods are the best solvers to use for this type of tuning problem. For problems with fewer than 20 variables, and a vast parameter space, both of our hybrid methods identified near-optimal solutions quickly and identified better solutions than any other strategy within the first 1000 function evaluations.

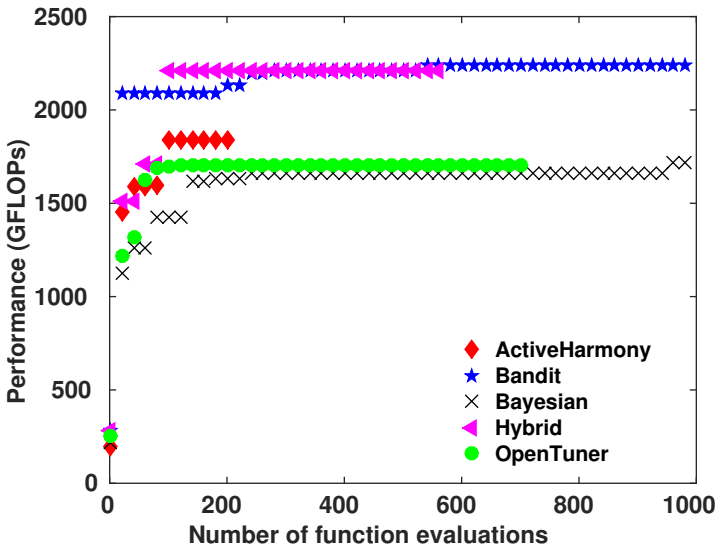


Figure 3.4: Comparison of the performance of different autotuners tuning dense matrix multiplication for 10000x10000 matrices on the Tesla P100 GPU

3.4.3 Tuning the GCC Compiler

The matrix multiplication problem addressed in the previous subsection involved 17 hyperparameters. In this subsection, we investigate how the proposed framework performs on more complex problems. The GCC compiler has over 300 optimization flags and tunable parameters and is challenging to tune. More than half of the tuning parameters are discrete choices, such as turning on an optimization flag, creating a complex parameter space that is nonsmooth and discontinuous. To improve the quality of the solutions returned by derivative-free optimization solvers and other autotuners, we performed a sensitivity analysis on all of the tuning parameters and optimization flags, for each algorithm that we compiled. We then experimented with the 50 most influential optimization flags and tuning parameters.

We conducted experiments using algorithms from the PolyBench 4.2.1 benchmark suite [179]. The algorithms contained in this library include different numerical methods: calculating sample covariance, BLAS routines, linear algebra kernels, stencil calculations, and more. All these codes are written in C. We compiled them using GCC version 4.8.5. We focused on three applications:

- doitgen, a kernel of multiresolution adaptive numerical scientific simulation;
- fdtd-2d, a simplified finite-difference time-domain method for 2D data; and
- syrk, a symmetric rank-k update.

In this study, we compare our proposed Bandit DFO and Hybrid DFO algorithms to other state-of-the-art autotuners, and report performance comparisons between them. Each autotuner was given 500 function evaluations. Below, we present profiles for the performance, measured as time, to execute the compiled code arising with each chosen set of hyperparameters. The results reported are the average execution time over three runs with different

starting points. For reference, we provide the performance of each algorithm compiled with the default GCC -O2 and -O3 optimization flags to demonstrate performance benefits for autotuning the GCC compiler.

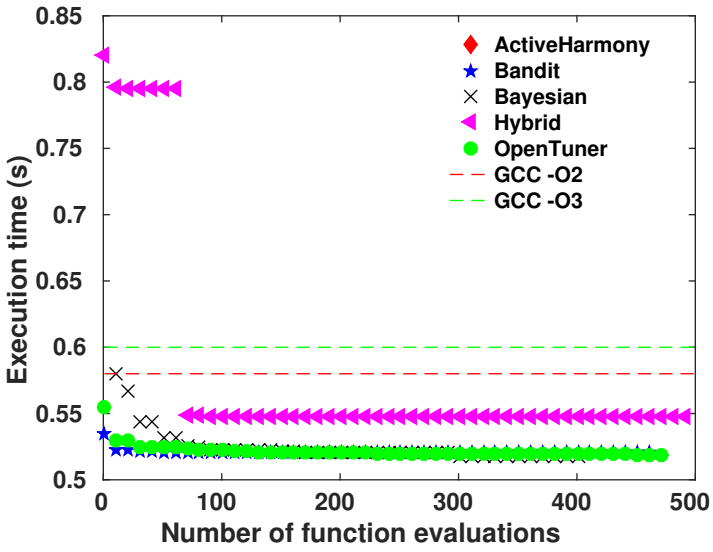


Figure 3.5: Performance of autotuners to tune compiling the doitgen algorithm

The performance of the tuners on the doitgen algorithm is shown in Figure 3.5, where we report the best found execution time on the y-axis against the number of iterations used on the x-axis. From these results, we see that all of the autotuners except ActiveHarmony found a better solution than the GCC default optimization flags, -O2 and -O3, within the first 20 iterations. ActiveHarmony, never identified a solution below 1.25 seconds within the first 500 iterations. All of the other solvers found comparable solutions that obtain a performance of 0.55 s within the first 100 iterations. OpenTuner and Bandit DFO located the best set of parameters in this experiment within 50 iterations.

In the case of tuning the FDTD-2D application, as shown in Figure 3.6, all of the autotuners determined parameters that are much faster than the default -O2 and -O3 optimization flags. Four of the methods found solutions that are more than twice as fast

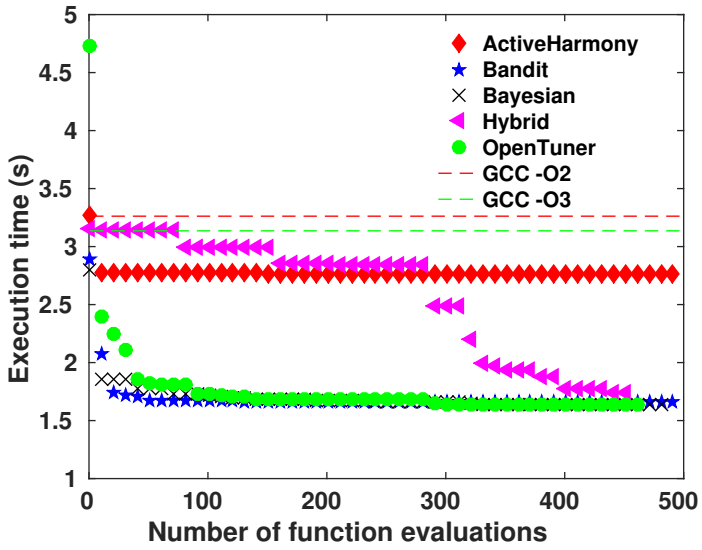


Figure 3.6: Performance of autotuners to tune compiling the FDTD-2D algorithm

as -O3. Bandit DFO identified the best observed solution within 60 function evaluations, while Bayesian Optimization and OpenTuner required over 100 iterations to achieve the same performance. In 400 iterations, Hybrid DFO converged to the same performance as the other autotuners. In Bandit DFO, glcDirect was called for 25 iterations before utilizing a local DFO solver. The performance of Bandit DFO suggests that the solution provided by DIRECT was critical to the algorithm locating the best known solution, as it converged after 30 iterations with a local solver.

For the results of the symmetric rank-k update algorithm shown in Figure 3.7, only two autotuners found parameter settings that outperformed the -O2 optimization flag. OpenTuner converged to a solution similar to the -O2 flag within 10 iterations, and never obtained a better solution. These results suggest that the -O2 parameter flag is a locally optimal solution that the local solvers used in OpenTuner were unable to escape from within 500 iterations. ActiveHarmony found a solution similar to the -O3 optimization flag after 200 iterations and then was unable to find a better solution. Hybrid DFO never converged to a

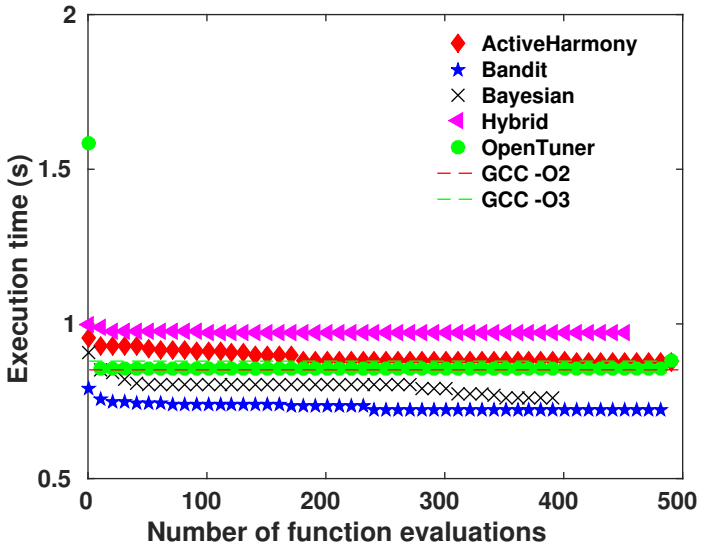


Figure 3.7: Performance of autotuners to tune compiling the symmetric rank-k update algorithm

solution comparable to the default -O3 compiler optimization. While Hybrid DFO typically performs well, problems like this suggest the need for utilizing a combination of local DFO solvers as in Bandit DFO. Both Bayesian Optimization and our Bandit DFO outperformed default compiler optimization flags. Within the first 10 calls of the syrk kernel, Bandit DFO obtained a solution that outperformed almost all of the other methods. Bayesian Optimization required 400 iterations to reach the same level of performance. We observe that the Bandit method found a near-optimal solution with fewer than 3% of the calls to the syrk kernel that are required by other techniques.

3.5 Conclusions

This paper investigates hybrid tuning algorithms for parameter tuning. While previous approaches rely on heuristics, or local direct search derivative-free optimization algorithms,

we propose hybridizing global DFO algorithms with local methods. We propose two hybrid methodologies, Bandit DFO and Hybrid DFO, that combine different DFO strategies to improve the rate at which tuners converge to an optimal solution.

We demonstrate that the two proposed hybrid algorithms outperform three state-of-the-art autotuners, ActiveHarmony, OpenTuner, and Bayesian Optimization with Gaussian Process models. Bandit DFO reduces the execution time of dense matrix multiplication by a factor of 1.4x compared to algorithms generated by other autotuners on a problem with a parameter space of $3.4x10^{11}$ combinations. We also demonstrate that the proposed methods are able to obtain superb solutions with less than 5% of the iterations required by other tuners when tuning the GCC compiler. Bandit DFO algorithm obtain the best observable solution faster than any other technique, in some cases, 20 times faster than other methods. In several examples, Bandit DFO was the only technique to obtain the best observable result within 500 iterations. Optimal parameters identified through tuning are 1.1x to 2x faster than default optimization flags used in GCC.

By combining global DFO strategies with local strategies, our hybrid algorithms identify the best observed parameters for the tuning applications that we present here. The proposed hybrid algorithms are generic and can tune problems of various sizes. To facilitate the development and use of autotuning software, we provided an open-source implementation of our Bandit DFO and Hybrid DFO algorithms for parameteric autotuning.

Chapter 4

Backward Stepwise Elimination

4.1 Introduction

Datasets may contain thousands of independent features, or nonlinear feature transformations. Feature selection is the problem of identifying a subset of features that succinctly and accurately relate a set of input observations to output measurements. We address the problem of feature selection by solving the best subset selection problem.

The best subset selection problem comes with three major challenges. First, it is NP-hard to solve with exact solution strategies [7]. When solving with branch-and-bound [68], mixed-integer optimization [24, 44], or exhaustive enumeration, optimal subset selection can become intractable for problems with a large number of features. Instead, heuristic techniques, such as forward stepwise selection (FSS), backward stepwise elimination (BSE) [58], or the lasso [166], are commonly used to identify near-optimal subsets for large instances [104]. Although heuristic approaches are significantly faster than exact methods, there are few studies that have investigated the accuracy of these methods.

The second challenge with subset selection is that it is not always obvious which features or nonlinear transformations should be considered when formulating the subset selection problem [69]. Including certain classes of features may generate complex models that perform well on training data, but perform poorly in practice. To overcome this challenge, feature selection can be performed on an extensive set of basis functions. However, as the computational cost for solving subset selection grows exponentially with the number of fea-

tures, it is not tractable to solve the subset selection problem to optimality for an arbitrary number of features.

The third challenge with subset selection is that, even when it is possible to solve the subset selection problem exactly, the mathematically optimal model may not be the best choice in practice. A recent study compared the accuracy of different subset selection strategies. In particular, Hastie et al. [83] compared the performance of FSS, the lasso [166], the relaxed lasso [119], and a mixed-integer formulation [24]. The comparisons did not consider BSE, thus leaving a gap in the understanding of this technique in comparison to other approaches.

We investigate the benefits of solving the best subset selection problem with a backward stepwise elimination algorithm. The contributions of this paper are:

1. We obtain an approximation guarantee for BSE using the concept of the supermodularity ratio. The derived guarantee provides a bound on the accuracy of backward stepwise elimination.
2. We propose a GPU parallel batched BSE algorithm that is a factor of 5x faster than a CPU implementation of BSE for a range of problem sizes.
3. We compare the accuracy of BSE and other state-of-the-art subset selection methodologies. We demonstrate that, for certain classes of problems, BSE generates models that are simpler and have less out-of-sample test error than the lasso or forward selection.

This paper is organized as follows. In Section 4.2, we review the literature related to best subset selection, stepwise selection, approximate submodularity, and supermodularity. In Section 4.3, we prove an approximation guarantee for BSE. In Section 4.4, we propose a batched GPU BSE algorithm, describe our implementation, and compare the performance of the proposed GPU algorithm against a CPU implementation. In Section 4.5, we compare

BSE against other popular subset selection techniques in terms of solution quality. We provide conclusions in Section 4.6.

4.2 Literature review

4.2.1 Best subset selection problem formulation

The best subset selection problem is defined as:

$$\min_{\boldsymbol{x}} \|\boldsymbol{y} - \mathbf{A}\boldsymbol{x}\|_2^2, \text{ subject to } \|\boldsymbol{x}\|_0 \leq k, \tag{4.1}$$

where $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$ when $m \geq n$, and k is the subset size. The ℓ_0 norm limits the number of nonzero coefficients and adds nonconvexity to an otherwise convex problem. Without the cardinality constraint, the problem

$$\min_{\boldsymbol{x}} \|\boldsymbol{y} - \mathbf{A}\boldsymbol{x}\|_2^2, \tag{4.2}$$

can be solved in closed form. A least-squares estimator of $\boldsymbol{x}$ can be found to solve $\boldsymbol{y} = \mathbf{A}\boldsymbol{x} + \epsilon$, where $\epsilon \in \mathbb{R}^m$. There are many techniques to solve the linear least squares problem, with QR factorization being one of the most commonly used. QR factorization involves decomposing a matrix $\mathbf{A} \in R^{m \times n}$ into the product of an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{m \times m}$ and an upper triangular matrix $\mathbf{R} \in \mathbb{R}^{n \times n}$:

$$\mathbf{Q}^T\mathbf{A} = \begin{pmatrix} \mathbf{R} \\ \mathbf{0} \end{pmatrix} \tag{4.3}$$

Let

$$\mathbf{Q}^T\boldsymbol{y} = \begin{pmatrix} \boldsymbol{y}_1 \\ \boldsymbol{w} \end{pmatrix} \tag{4.4}$$

where $\boldsymbol{y}_1 \in \mathbb{R}^n$ and $\boldsymbol{w} \in \mathbb{R}^{m-n}$. The least squares estimator is obtained by solving the following optimization problem

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \|\boldsymbol{y} - \mathbf{A}\boldsymbol{x}\|_2^2 = \arg\min_{\boldsymbol{x}} \|\mathbf{Q}^T(\boldsymbol{y} - \mathbf{A}\boldsymbol{x})\|_2^2. \tag{4.5}$$

From this, the residual sum of squares (RSS) can be calculated from

$$RSS = \|\boldsymbol{y} - \mathbf{A}\hat{\boldsymbol{x}}\|_2^2 = \|\mathbf{Q}^T(\boldsymbol{y} - \hat{\boldsymbol{x}})\|_2^2 = \|\boldsymbol{w}\|_2^2. \tag{4.6}$$

If QR factorization is used, the residual sum of squares is calculated from the Euclidean norm of $\mathbf{Q}^T\boldsymbol{y}$ for the vector of values from $n+1$ to m.

4.2.2 Stepwise selection and elimination

A common technique for solving (4.1) involves selecting or eliminating variables, in a stepwise fashion. Forward stepwise selection initially generates a model that minimizes RSS by selecting a single variable. Then, in each subsequent iteration, a new variable is included in the solution until $\|\boldsymbol{x}\|_0 = k$. In every iteration, a new model is obtained by identifying the variable that minimizes RSS when added to the previously obtained model. Forward stepwise selection (FSS) is a greedy selection algorithm, which has a provable worst performance for certain classes of problems [126]. Forward selection can also be used for problems when the optimal subset size is not known *a priori*. Stopping rules for FSS aim to find a balance between accuracy and model complexity [22].

Backward stepwise elimination (BSE) starts from the standard least squares solution and removes one feature at a time until the cardinality constraint is satisfied. Given the initial least squares solution $\boldsymbol{x}_0$, the error for the model after s iterations and the corresponding subset $\boldsymbol{x}_s$ are obtained via factorization or QR downdating [28]. QR downdating refers to updating the solution to the linear least squares problem when a column or a row is removed from $\mathbf{A}$. For subset selection, a model with k columns is selected, then RSS is calculated, where $\boldsymbol{y}_2 \in \mathbb{R}^{m-k}$. QR downdating reduces the number of floating point operations by removing a column from $\mathbf{R}_s$ and updating $\mathbf{Q}$ to maintain an upper triangular structure in $\mathbf{R}_{s-1}$ without having to perform a QR factorization at every iteration. When a column is removed from $\mathbf{R}_{s-1}$, the upper triangular structure is only destroyed in the columns to

the right of the deleted column. As the computational complexity of QR factorization is $O(2mn^2)$, in many cases, the complexity of the update is significantly less than the cost of factorizing $\mathbf{A}_{s-1}$. The sum of squared errors is computed by left multiplying $\mathbf{Q}^T$ to restore the upper triangular structure $\mathbf{R}_{s-1}$ with $\boldsymbol{y}_2$. This procedure is repeated to calculate the best $i = n, \ldots, n_{\min}$ models, where $n_{\min} \in [1, k]$. In each iteration, after the best model has been identified, all suboptimal solutions are discarded, and the next iteration begins. This algorithm is outlined in Algorithm 1.

Algorithm 1 Generic Backward Subset Elimination Algorithm

Given a set of data points $\boldsymbol{x}_{ij}, \boldsymbol{y}_i$ for $i = 1, ..., m$, $j = 1, ..., n$
Generate a set of basis functions from input features $\mathbf{A}$
procedure BackwardElimination($\mathbf{A}$)
    $\mathbf{Q}_n\mathbf{R}_n \leftarrow \mathbf{A}$
    for $k = n - 1, ..., 0$ **do**
        for $h = 1, ..., k$ **do**
            $\mathbf{A}_{k,h} \leftarrow \mathbf{R}_{k+1}$
            $\mathbf{Q}_{k,h}\mathbf{R}_{k,h} \leftarrow QR(\mathbf{A}_{k,h}(h : m, h : n))$
            $\boldsymbol{w}_{k,h} \leftarrow \mathbf{Q}_{k,h}^T \boldsymbol{w}_{k+1}$
            $\mathbf{RSS}_{k,h} \leftarrow \|\boldsymbol{w}_{k,h}(k + 1 : m)\|_2^2$
        end for
        $i \leftarrow \arg\max_h \boldsymbol{w}_{k,h}$
        $\boldsymbol{w}_k \leftarrow \boldsymbol{w}_{k,i}$
        $\mathbf{A}_k \leftarrow \mathbf{A}_{k,i}$
    end for
end procedure

Both stepwise techniques have been extensively used for the last fifty years [67]. In most

cases, BSE requires more floating point operations than FSS. However, the accuracy of BSE has been observed to be better than FSS for certain classes of problems [117, 43]. While some have argued that neither approach should be used [145], for problems with millions of observations and thousands of features, stepwise approaches quickly generate approximately accurate and sparse solutions.

FSS and BSE are heuristic hill climbing strategies that obtain locally optimal solutions to the subset selection problem. As both methods require fewer computations than exact strategies, researchers have investigated if any guarantees exist for these methods. Several authors have proven statistical bounds on the accuracy of FSS [52, 126]. Using the notion of the submodularity ratio, it is possible to obtain a worst-case bound on the performance of FSS. For BSE, Couvreur and Bresler [43] have proven that, under certain conditions, BSE identifies an optimal subset. Unfortunately, checking the conditions in [43] requires the solution of an NP-hard problem.

4.2.3 Submodularity and supermodularity

A function f that maps a set to a real number is called submodular if it satisfies the following property:

$$f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T), \tag{4.7}$$

for $S \subset T$ and $\{v\} \subset T \setminus S$. The results of Nemhauser et al. [126] prove that the greedy algorithm achieves a $(1-1/e)$-approximation for the maximization of any monotone, submodular set function over a cardinality constraint. Here, e is the base of the natural logarithm. This approximation result provides a lower bound on the performance of greedy algorithms for solving NP-hard problems subject to cardinality constraints. However, subset selection does not involve a submodular objective function. To develop an approximation guarantee for subset selection, the work of [52] defines the submodularity ratio as a way to measure how

close a function is to being submodular:

$$\gamma_{U,k}(f) = \min_{L \subset U, S: |S| \leq k, S \cap L = \emptyset} \frac{\sum_{x \in S} f(L \cup \{x\}) - f(L)}{f(L \cup S) - f(L)}, \tag{4.8}$$

where f is a set function, $L \subset U$ and $S \cap L = \emptyset$. The submodularity ratio is a function of the maximum subset size k, and the set U. It reflects how much the value of f increases by adding any subset S of size k to L, compared to the benefit of $f(S \cup L)$. If the function f is submodular, then the submodularity ratio is defined to be 1, otherwise if $\gamma < 1$, the function is defined as weakly submodular. The authors of [52] prove that FSS has a worst-case approximation guarantee of $1 - \exp(-\gamma) \cdot OPT$, where OPT is the R^2 of the optimal best subset solution. For $\gamma = 1$, the guarantee in [52] recovers the guarantee of Nemhauser et al.; the bound is loose as γ approaches zero.

A function f is supermodular if $-f$ is submodular. Several authors have defined a supermodularity ratio [112, 98, 148]. Inspired by the work of [148], we define the following supermodularity ratio:

$$\beta_{U,k}(f) \geq \frac{\sum_{x \in S} f(L \setminus \{x\}) - f(L)}{f(L \setminus S) - f(L)}, \tag{4.9}$$

where $\beta_{U,k} \in [1, k]$ is selected as the maximum value for each combination of $S, L \subseteq U$. Like the submodularity ratio, the supermodularity ratio captures how close a function is to being supermodular.

4.3 Algorithmic Analysis of BSE

While there exist approximation guarantees for forward selection, no such bound is currently known for backward stepwise elimination. To determine such a bound, we use the concept of the supermodularity ratio.

Let f be a nonnegative monotonically increasing set function. The problem we seek to solve is:

$$\max_{S} f(S), \text{ subject to } \|S\|_0 \leq k. \tag{4.10}$$

Our theoretical contribution is an approximation guarantee on the performance of backward stepwise elimination.

Theorem 1. *Let f be a nonnegative, monotonically increasing set function, OPT be the maximum value of f possible for a set of size k, and k^* be the size of the subset for OPT. Then the set selected by BSE, S_{n-k}^{BSE}, has the following approximation guarantee:*

$$f(S_{n-k}^{BSE}) \geq \left(1 - \frac{\beta}{k^*}\right)^{n-k^*} \cdot OPT. \tag{4.11}$$

Proof. Let S_0^B be the initial set of all variables considered, and S_{n-k}^* be an optimal set of k variables that has a value of OPT. Let S_i^B be the set of variables that remain in S after i iterations of BSE. We begin by rearranging the supermodularity ratio to ensure that the numerator and denominator in (4.9) are both positive:

$$\sum_{x_j \in S_i^B} \left(f(S_i^B) - f(S_i^B \setminus \{x_j\})\right) \leq \beta \left(f(S_i^B) - f(S_i^B \setminus S_i^B)\right). \tag{4.12}$$

In every iteration of BSE, $\hat{x}$ is selected to minimize $f(S_i^B) - f(S_i^B \setminus \{\hat{x}\})$. As the minimum size of S_i^B is $|S_i^B| \geq k$ and $\sum_i^n x \geq n \cdot x_{\min}$, we have:

$$k^* \left(f(S_i^B) - f(S_{i+1}^B)\right) \leq |S_i^B| \left(f(S_i^B) - f(S_{i+1}^B)\right) \leq \beta f(S_i^B). \tag{4.13}$$

Letting $A(i)$ be the loss in f in iteration i, $A(i) = f(S_{i-1}^B) - f(S_i^B)$. Let $f(S_0^B)$ be the value of f when all variables in the set are included. Then $\sum_{j=1}^{i} A(j) = f(S_0^B) - f(S_i^B)$ extends from the definition of $A(i)$. Rewriting (4.13) in terms of $A(\cdot)$, we get:

$$A(i+1) \leq \frac{\beta}{k} \left(f(S_0^B) - \sum_{j=1}^{i} A(j)\right). \tag{4.14}$$

Using the inequality above, we will prove by induction that:

$$f(S_0^B) - \sum_{j=1}^{t} A(j) \geq f(S_0^B)\left(1 - \frac{\beta}{k}\right)^{n-k} \geq \left(1 - \frac{\beta}{k}\right)^{n-k} \cdot OPT \tag{4.15}$$

For $t = 0$, the inequality is trivial. Assume that the inequality holds for t iterations. Then, for iteration $t + 1$:

$$\begin{aligned}
f(S_0^B) - \sum_{j=1}^{t+1} A(j) &= f(S_0^B) - \sum_{j=1}^{t} A(j) - A(t+1) \\
&\geq f(S_0^B) - \sum_{j=1}^{t} A(j) - \frac{\beta}{k}\left(f(S_0^B) - \sum_{j=1}^{t} A(j)\right) \\
&\geq \left(f(S_0^B) - \sum_{j=1}^{t} A(j)\right)\left(1 - \frac{\beta}{k}\right) \\
&\geq f(S_0^B)\left(1 - \frac{\beta}{k}\right)^{t+1} \\
&\geq f(S_0^B)\left(1 - \frac{\beta}{k}\right)^{n-k}
\end{aligned}$$

where $t + 1$ is less than or equal to $n - k$. Finally, $f(S_0^B) \geq OPT$ and from the definition of $f(S_{n-k^*}^{BSE}) = f(S_0^B) - \sum_{j=1}^{n-k} A(j)$:

$$f(S_{n-k}^{BSE}) \geq \left(1 - \frac{\beta}{k}\right)^{n-k} \cdot OPT. \tag{4.16}$$

This completes the proof for the approximation guarantee. □

We apply this theorem to the best subset selection problem by defining $f(S) = R_S^2$. When $\beta = 1$, our approximation is the tightest, and deteriorates until $\beta = k$. This implies that the proposed guarantee is stronger for functions that are closer to supermodular, similar to the submodularity ratio for submodular functions. Additionally, the proposed bound is stronger as k approaches n, where in the case that $k = n$, BSE returns the linear least squares solution.

4.4 A batched GPU algorithm for BSE

One major criticism against BSE is that it is computationally expensive. To address this shortcoming, in this section, we develop a parallel BSE algorithm using batched GPU computing.

4.4.1 Background

To reduce the computational time of BSE, we parallelized the QR downdate operations in each iteration of Algorithm 1. Unfortunately, in every iteration, each downdate task is unique. As a result, it cannot be accelerated with a data-parallel framework. In particular, the problem size and batch size decrease in each iteration, and every downdate requires a different number of matrix update operations. Instead of a data-parallel approach, we parallelized tasks with batched GPU computing.

GPUs are powerful accelerators that are designed for single instruction multiple data parallelism—not task-level parallelism. GPU hardware is designed for rendering graphics and performing the same set of operations on different sets of data. There are many cases where thousands of small, independent problems need to be solved. To take advantage of GPU hardware for scientific computing, "batching" is a technique that solves groups of problems in parallel [79, 42]. While algorithms designed for batched computing do not fully utilize the hardware, batched methods have been observed to be a factor of 2x faster than optimized CPU kernels for performing the same set of instructions [56].

Despite the clear need to solve problems in batches, developing software to execute task-level parallelism on a GPU efficiently is challenging. To fill this gap, two batched dense linear algebra libraries have been developed. In CUBLAS, NVIDIA developers have created a set of batched basic linear algebra subroutines (BLAS) and batched kernels for QR factorization, LU factorization, and matrix-matrix multiplication [42]. The Innovative

Computing Laboratory developed MAGMA, and implemented efficient batched BLAS routines to accelerate batched linear algebra kernels [79]. MAGMA has demonstrated that, with proper algorithm-specific optimizations, it is possible to develop algorithms that are twice as fast as CUBLAS, for problems with a large batch size. Additionally, for small- to medium-sized problems, batched BLAS approaches are reported as 2-3x faster for batched matrix-multiplication compared to traditional GPU code [79].

4.4.2 Batched BSE

We employed the batched QR factorization routine available in the MAGMA 2.5.0 library [93]. While batched QR factorization is the most time consuming portion of the backwards stepwise algorithm, a BSE algorithm also needs to perform downdates on the output $\boldsymbol{y}$ to calculate the sum of squared errors for every problem in a batch. Unfortunately, MAGMA does not have a batched implementation to perform QR downdates. In LAPACK, this functionality corresponds to the routine DORMQR, which uses $\mathbf{Q}$ generated from DGEQRF and calculates $\mathbf{Q}^T\boldsymbol{y}$.

We augmented the DGEQRF routine to include the update operation on $\boldsymbol{y}$. We modified the batched routine to update $\boldsymbol{y}$ when the rest of the matrix $\mathbf{A}$ is updated. When $\boldsymbol{y}$ was a vector, updating $\boldsymbol{y}$ added a negligible amount of time. We conducted experiments, and observed that the computational time of the modified code did not increase compared to that of the original batched DGEQRF code.

The batched BSE algorithm computes a solution to the linear least squares problem. Then, in parallel, the algorithm removes different single features from the linear least squares solution. Each task then downdates $\boldsymbol{y}$ to calculate the updated sum of squared errors. After factorizing and updating $\boldsymbol{y}$ for the removal of each candidate variable, the column with the smallest change in SSE is removed. This process is repeated until terminating at a predefined

minimum matrix size.

The backwards selection algorithm relies on the QR factorization kernel in MAGMA for tall and skinny matrices. To optimize the performance of this routine, we performed parameter tuning on the block size parameter in MAGMA. From previous work [151], it was observed that varying the block size parameter in MAGMA has a significant impact on performance. We discovered that changing the block size to 16, from a default value of 32 improved the performance of the batched kernel for problem sizes of interest.

By definition, a batch is made up of problems of the same size. In each iteration, we perform matrix updates that operate on a different number of columns ranging from zero to $n - s_i$ columns, where n is the number of columns in the matrix and s_i is the current iteration. As a result, for every task in the same iteration, the number of operations in each downdate operation is different, depending on which column is removed. If the feature removed from $\mathbf{R}$ is the furthest to the right, no work is needed to downdate the solution. However, if the first column is removed, the entire matrix needs to be downdated to restore the upper triangular structure of $\mathbf{R}$. This uneven distribution of work creates a batch size of one, where all jobs require different computations. To make BSE amenable to batched computing, we decided to perform downdate operations for every feature as if the entire matrix is to be downdated. By assuming that all problems are the same size, we greatly increase the total number of computations in every batch. This design choice allows us to set the batch size equivalent to the number of features that are candidates to remove in each iteration, i.e., $n - s_i$. Even though doing so increases the total count of operations, our early computational experimentation demonstrated that the proposed batch methodology is a reasonable option. In particular, the proposed algorithm is faster than a sequential CPU BSE implementation and a GPU BSE implementation. An outline of the algorithm listed above is detailed in Algorithm 2.

Algorithm 2 Batched Backward Subset Elimination Algorithm

Given a set of data points $\boldsymbol{x}_{ij}, \boldsymbol{y}_i$ for $i = 1, ..., m$, $j = 1, ...n$

Generate a set of basis functions from input features $\mathbf{A}$

procedure BatchBackwardElimination($\mathbf{A}$)

    $\mathbf{Q}_n\mathbf{R}_n \leftarrow \mathbf{A}$

    for $k = n - 1, ..., 0$ **do**

        $\mathbf{A}_k \leftarrow \mathbf{R}_{k+1}$

        $\mathbf{Q}_k\mathbf{R}_k \leftarrow QR(\mathbf{A}_k)$

        $\boldsymbol{w}_k \leftarrow \mathbf{Q}_k^T \boldsymbol{w}_{k+1}$

        $\mathbf{RSS}_k \leftarrow \|\boldsymbol{w}_k(k+1:m)\|_2^2$

        $i \leftarrow \arg\max_k \boldsymbol{w}_k$

        $\boldsymbol{w}_k \leftarrow \boldsymbol{w}_{k,i}$

        $\mathbf{A}_k \leftarrow \mathbf{A}_{k,i}$

    end for

end procedure

4.4.3 Computational results

We conducted experiments on a machine running CentOS7, with an Intel Xeon E5-1630 at 3.7 GHz and 8 GB of RAM. The machine was equipped with a NVIDIA Tesla K40 GPU with 15 streaming multiprocessors, 12 GB of RAM, and a peak memory bandwidth of 288 GB/s. The algorithms were compiled with the NVCC CUDA 9.1 compiler, using the -03 optimization flag. We generated subset selection problems with randomly generated values between zero and one. We compare the computational time to solve the best subset selection problem for problems with $m = 500 - 1000$ over a range of $n = 200 - 600$. We consider problems where the number of rows is larger than the number of columns.

For each problem size, we generated 10 problem instances and calculated the average execution time. We utilized the MAGMA-2.5.0 [93] library to perform batched least squares calculations, with the modification to the DGEQRF routine that we detailed above to perform QR downdating. We compared the proposed batched GPU algorithm against a CPU implementation of BSE that relies on LAPACK [9] to perform factorization and QR downdating.

As seen in Figure 4.1, the parallel backward elimination algorithm is 2-5x faster than the CPU implementation. In the figure, we report the GPU speedup as a function of the number of columns, for three matrix sizes. We observe that the speedup levels off as the number of columns, or equivalently the batch size, increases. A leveling off of performance is indicative that the computing resources are completely saturated.

The speedup obtained when the number of rows is increased is not as significant as when the number of columns is increased. As the computational complexity of QR factorization scales with the square of the number of columns and linearly with the number of rows, our speedups are in line with the computational complexity of the underlying algorithm.

To reinforce the observation that the speedup for BSE was limited by computational efficiency of the computing resources, we investigated the performance of the algorithm as a function of the batch size. Figure 4.2 displays the execution time of the CPU and GPU BSE algorithms as a function of batch size. In every iteration of BSE, the batch size was decreased by one. From Figure 4.2, we see that the benefits of batched GPU computing decrease as the problem size decreases. For batch sizes equal to 600, the GPU outperforms the CPU by a factor of 5x. For large batch sizes, above 300, the execution time increases linearly as the problem size increases. A linear relationship between execution time and problem size suggests that performance is limited by a computational bottleneck. Even though the GPU outperforms the CPU for large batch sizes, the speedup decreases to one

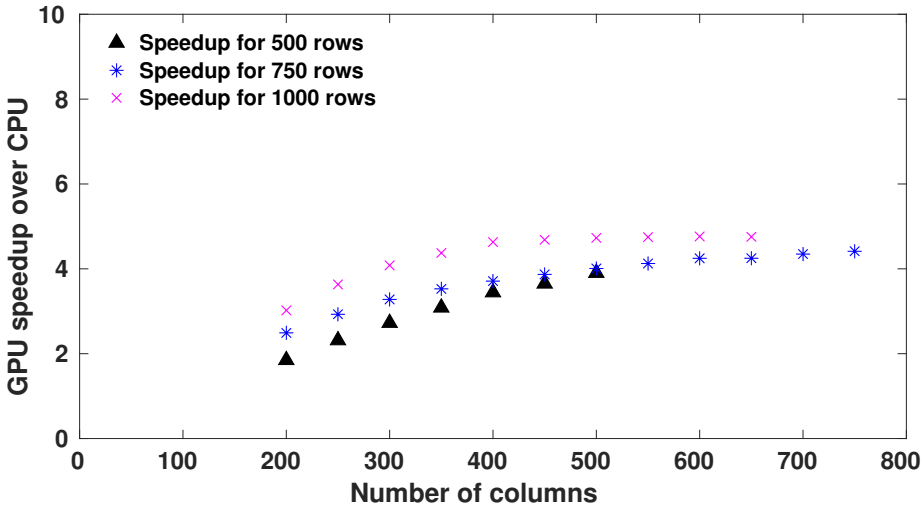


Figure 4.1: Speedup as a function of problem size for the backward stepwise elimination algorithm

around a batch size of 50. The CPU is faster than the batched GPU algorithm for small problems. For small problems, the overhead of transferring data to the GPU outweighs the benefits of batched computing.

4.5 Accuracy of backward stepwise elimination

4.5.1 Background

Recently, several articles have been published on the topic of best subset selection. With advances in integer programming solvers, researchers have investigated this problem with mixed-integer programming techniques [24, 44]. However, in the statistics community, several have postulated that it may not be worthwhile to solve this problem to optimality on training data [83, 167]. Instead, the use of heuristic approaches like the lasso and forward selection have been investigated and found to perform well for various problems [83]. In

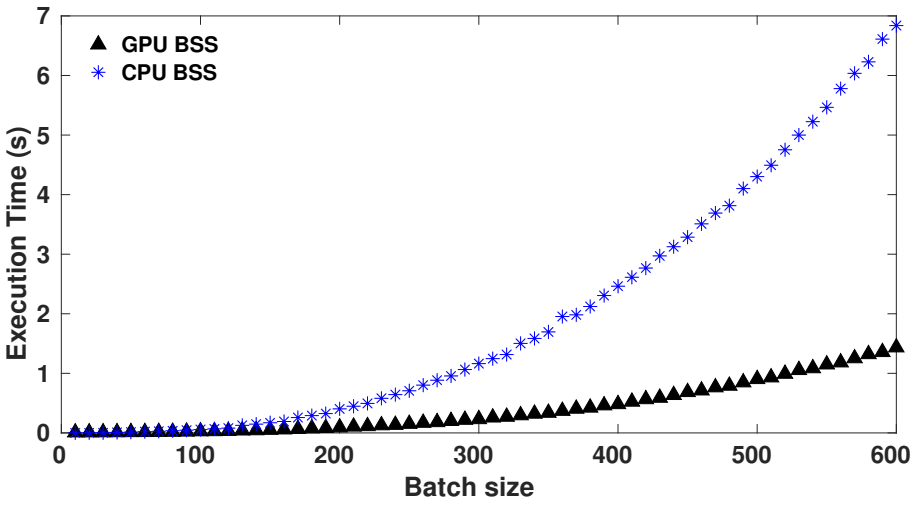


Figure 4.2: Execution time as a function of batch size for the backward stepwise selection algorithm for a problem with 1000 rows and 600 columns with a variable batch size between one and 600

the work of Hastie et al. [83], both the execution time and several out-of-sample statistical metrics are used to compare the lasso, a mixed-integer programming formulation, forward selection, and the relaxed lasso. They discovered that each of the methods obtained the best solution under different problem sizes and data characteristics. In terms of computational time, the mixed-integer programming formulation was the most computationally expensive for all problems considered.

The work of Hastie et al. raised two questions that we investigate in this paper. First, the examples formulated in their work sought to identify a sparse algebraic representation for models with five variables in the true model. However, in practice, modeling complex systems may require complex non-linear equations with more terms. Second, while forward and backward selection have been compared empirically in the literature, we are interested in determining when BSE should be used for subset selection. To facilitate a comparison

between these methods, we performed experiments with four techniques:

1. the proposed batched BSE algorithm,
2. forward selection in the R best-subset package,
3. the lasso in the R best-subset package,
4. the relaxed lasso in the R best-subset package.

Despite recent advances in solving mixed-integer problems, for problems of sufficient size, solving the best subset selection problem exactly is still costly. We do not include results for the mixed-integer formulation as the approximate best subset solutions obtained from preliminary experiments were comparable to forward selection.

4.5.2 Experimental setup

In this section, we make use of the notation proposed in Hastie et al. [83]. Our experiments followed a similar procedure to those presented in the Hastie et al. paper, and were conducted on the same machine as in the previous section. Data in our experiments were drawn from distributions that were defined by several parameters. Our matrices were generated by defining a problem size (m, n), a sparsity level s, to indicate the number of nonzeros in the model, and a beta-type, to create a sparsity pattern. Additionally, ρ is used to control the correlation level between variables when generating input data, and a signal-to-noise-ratio (SNR) term was used to control the level of noise in the data. Matrices were generated from a true model parameterized by ρ and s. A response vector $\boldsymbol{y}$ was also drawn by sampling points from the true model while adding noise that satisfied a specified SNR.

To compare approaches, several test metrics were evaluated: relative risk, relative test error, proportion of variance explained, and the number of nonzeros in the chosen model.

As studied in Hastie et al., relative risk (RR) is a measure of predictive performance.

$$RR(\hat{\beta}) = \frac{(\hat{\beta} - \beta_0)^T \Sigma (\hat{\beta} - \beta_0)}{\beta_0^T \Sigma \beta_0} \tag{4.17}$$

Here, $\hat{\beta}$ is the vector of coefficients selected from regression, β_0 is the vector of true coefficients that are used to generate the data, and Σ represents the correlation between the predictor variables. A perfect RR score for relative risk is zero, corresponding to $\hat{\beta} = \beta_0$. A bad score corresponds to one. Relative test error (RTE) is an out-of-sample procedure for measuring accuracy, which measures the expected test error relative to the Bayes error rate:

$$RTE = \frac{(\hat{\beta} - \beta_0)^T \Sigma (\hat{\beta} - \beta_0) + \sigma^2}{\sigma^2} \tag{4.18}$$

A perfect RTE score is one, while a score of zero corresponds to $\hat{\beta} = 0$. In this formula, σ^2 is the variance used to generate the matrices while satisfying a predetermined SNR. Proportion of variance explained is the amount of variance explained by the proposed model in the output variable y_0:

$$PVE = 1 - \frac{(\hat{\beta} - \beta_0)^T \Sigma (\hat{\beta} - \beta_0) + \sigma^2}{\beta_0^T \Sigma \beta_0 + \sigma^2} \tag{4.19}$$

If the true model is selected, PVE equals $\frac{SNR}{1+SNR}$, while a null model has a score of zero. The last metric considered is the number of nonzero coefficients selected. In general, sparser models generalize better to validation data.

To compare BSE against other subset selection strategies, we conducted experiments with matrices of size $m = 500$, $n = 100$, and $s = 5$. We were also interested in determining which methods are better suited for developing more complex models. We consider s over a range of 10 to 70 in multiples of 20. Experiments for all problem types were conducted over $SNR \in [0.05, 0.09, 0.14, 0.25, 0.42, 0.71, 1.22, 2.07, 3.52, 6]$.

In the work of Hastie et al., multiple methods were used to generate matrices. We used beta-type 2, where β_0 has the first s parameters equivalent to one, with the rest set to zero.

Experiments were conducted with ρ equivalent to either 0 or 0.35. All values reported below are an average over five repetitions. For each technique, a solution path was generated for every SNR considered. The results reported below are from the models that minimized the desired test metric from each solution path.

4.5.3 Computational results

Figures 4.3 and 4.4 relate SNR to the accuracy metrics for different correlation levels. The uncorrelated case was unique from the other cases that were observed. BSE and FSS performs differently when $SNR < 0.16$. In particular, BSE in the low SNR cases outperforms all other methods in regards to RR and RTE. For $SNR > 0.16$ all of the methods except for the lasso converge to low error solutions. The lasso selects denser models than all of the other methods, selecting a 25-term model as opposed to a five-term model.

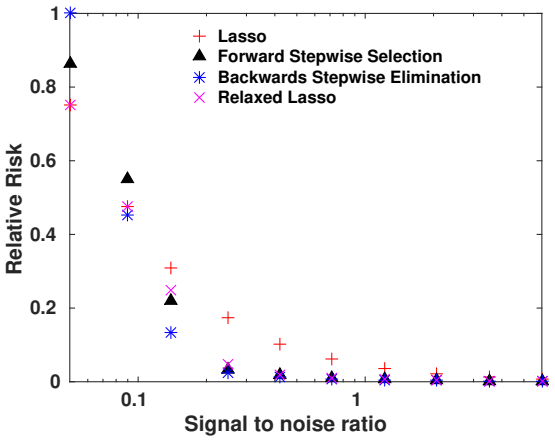
The results suggest that BSE outperforms the other methods at $SNR < 0.16$. The relaxed lasso and lasso both select denser solutions than BSE for these problems. BSE outperforms FSS because FSS selects several variables in early iterations that hinder its overall performance as k increases. For this case of noisy data with no correlation, BSE selects a sparser model than the relaxed lasso, leading to a smaller RTE.

At a larger correlation of $\rho = 0.35$, the advantage demonstrated by BSE at the low SNR regime vanishes. BSE and FSS perform similarly except for small deviations in RTE observed at $SNR = 0.42$. All of the methods converge to a similar RTE around $SNR = 0.71$, except for the lasso. The lasso selects a denser solution than all of the other methods, and does not converge to the RTE obtained by the other methods. The relaxed lasso does not have this problem as it manipulates a second tuning parameter γ to control the aggressiveness of the relaxed lasso to shift its performance from that of the lasso to that of

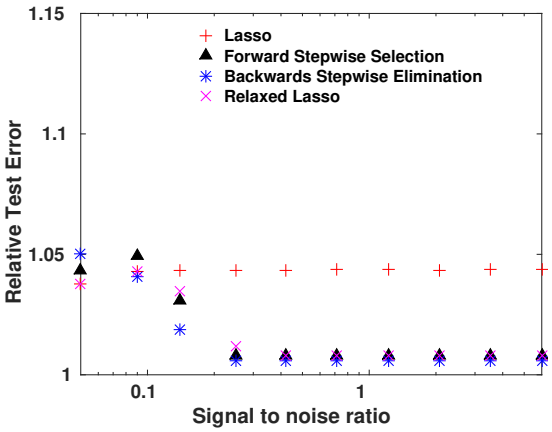
best subset and forward selection. The results demonstrate that either BSE or the relaxed lasso are the best methods for problems of this size. The choice of which method to select depends on the correlation in the underlying data. The correlation of the features affects the critical transition value after which BSE, best subset, and FSS outperform the lasso and are competitive with the relaxed lasso method. BSE has a lower RTE than the relaxed lasso only in the case of $\rho = 0$.

In Figure 4.5, we report results relating RTE to s. RTE is affected most by a change in the number of nonzeros in the model. Similar to the results of Hastie et al., models generated have a critical transition value at which point the RTE of BSE and FSS decreases below that of the lasso. The performance of the lasso is worse than all of the other methods above the critical transition value, while that of the relaxed lasso is similar to that of the stepwise methods. Unlike in the $s = 5$ case, in all of the results, the relaxed lasso outperforms BSE for SNR less than the critical transition value. The most notable result from this study is that, in certain cases above the critical transition value, BSE and FSS outperform the relaxed lasso. For $s = 30$, BSE outperforms FSS and the relaxed lasso for $SNR = 1.22$. We also investigated whether the RTE converges for all methods if the SNR value is increased beyond six. At larger SNR values approaching 20, BSE still outperforms FSS and the relaxed lasso. From this comparison, it appears that, in the case of low correlation in the input data and regardless of how large the underlying model is, BSE is competitive with other methods at any SNR. The relaxed lasso and FSS also generate accurate models for problems of this size.

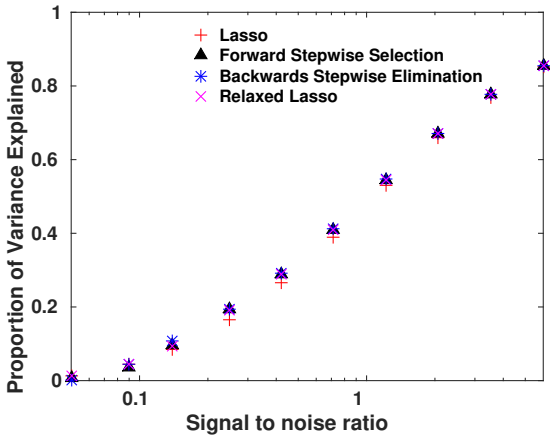
Depending on the problem structure, different subset selection strategies are optimal. We expected that BSE would outperform forward selection when the number of terms in the true model approaches n as suggested by the proposed approximation guarantee in Section 4.3. This trend was observed for $\rho = 0$. Overall, the best technique to use depends on the



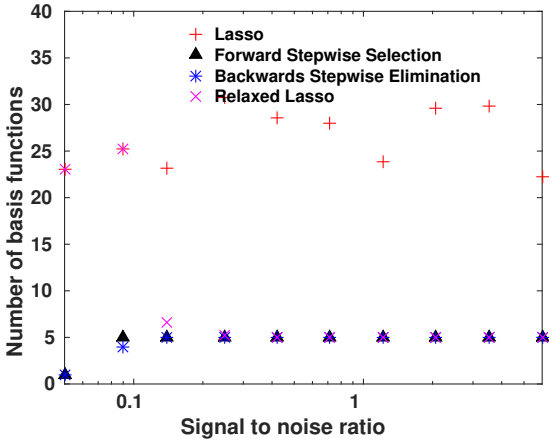
(a) Relative risk as a function of SNR



(b) Relative test error as a function of SNR

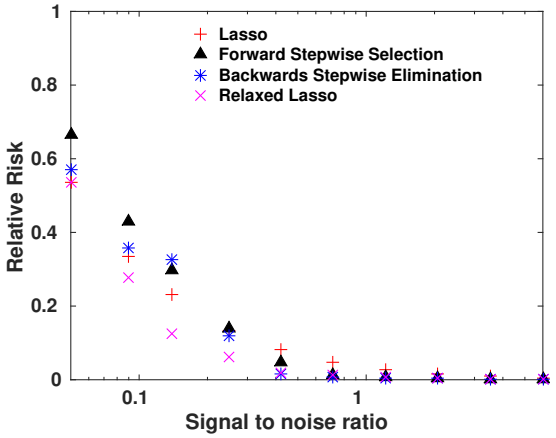


(c) Proportion of variance explained as a function of SNR

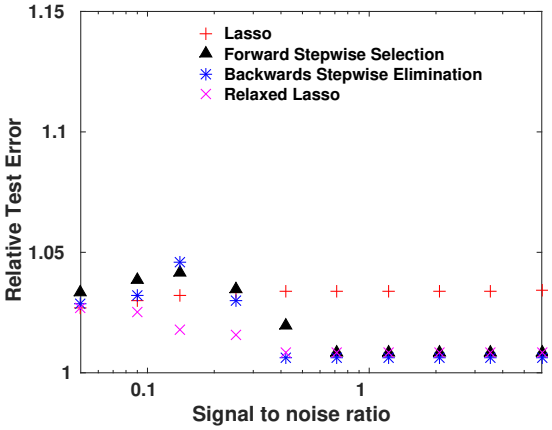


(d) Number of nonzero coefficients as a function of SNR

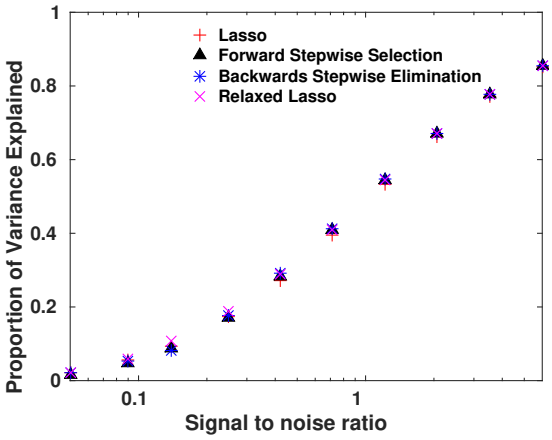
Figure 4.3: Four accuracy metrics for the performance of different subset selection techniques for $\rho = 0$



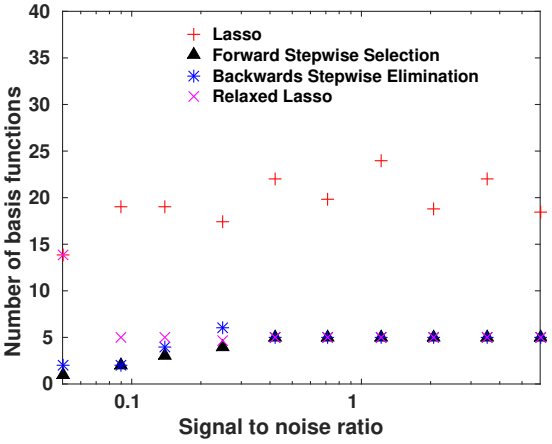
(a) Relative risk as a function of SNR



(b) Relative test error as a function of SNR

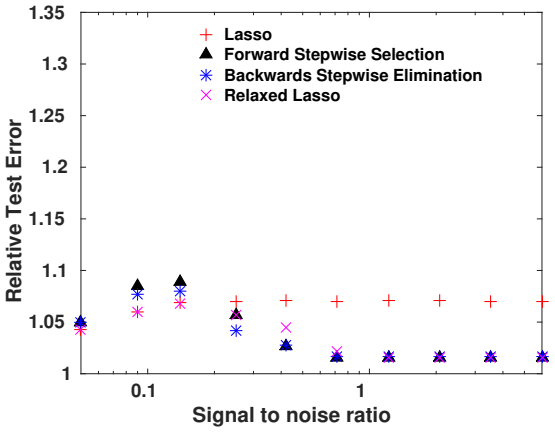


(c) Proportion of variance explained as a function of SNR

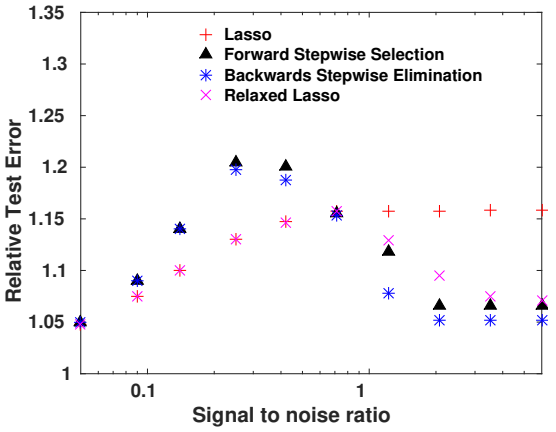


(d) Number of nonzero coefficients as a function of SNR

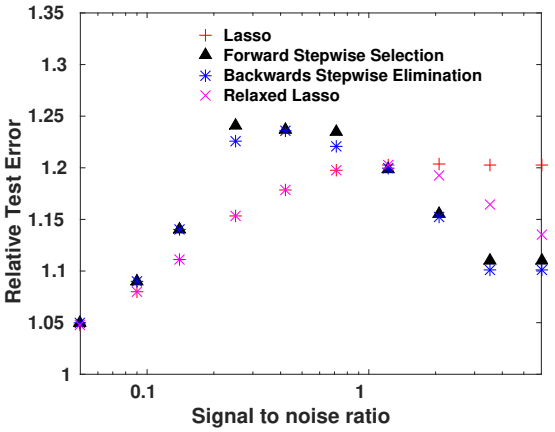
Figure 4.4: Four accuracy metrics for the performance of different subset selection techniques for $\rho = 0.35$



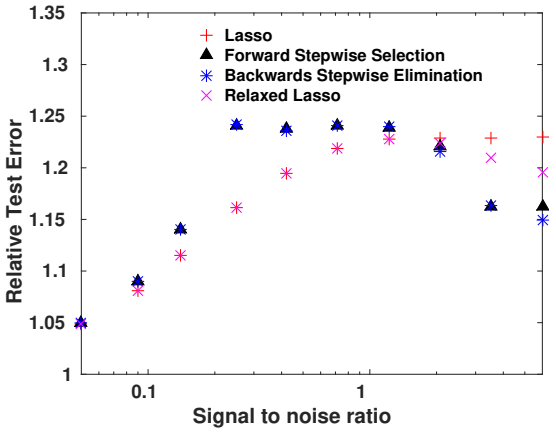
(a) RTE as a function of SNR for $s = 10$



(b) RTE as a function of SNR for $s = 30$



(c) RTE as a function of SNR for $s = 50$



(d) RTE as a function of SNR for $s = 70$

Figure 4.5: Four accuracy metrics for the performance of different subset selection techniques when the number of nonzero coefficients in the real model changes for problems with $\rho = 0$

underlying data. For certain classes of problems, especially those that are uncorrelated, BSE produces an accurate and sparse model.

4.6 Conclusions

We investigated using backward stepwise elimination to solve the subset selection problem. Using the concept of the supermodularity ratio, we obtained an approximation guarantee for backward stepwise elimination. Our computational results demonstrate that the performance of backward stepwise elimination is dependent on the difference between n and k, and more unexpectedly, the supermodularity ratio. We developed a GPU parallel batched BSE algorithm. This algorithm reduces the execution time of solving the subset selection problem for matrices with 1000 rows and 600 columns by a factor of 5x.

We demonstrated that BSE performs as well as other state-of-the-art subset selection strategies that are commonly employed in practice. For certain problems at SNR below 0.5, BSE generated sparser models and achieved a lower relative test error than forward selection and the lasso. Results demonstrated that BSE also achieved a lower relative test error than the relaxed lasso, the lasso, or forward stepwise selection for problems with no correlation and for signal to noise ratios above zero.

Our primary conclusion is that BSE is a technique that should be considered by practitioners who want to develop sparse and accurate models.

Chapter 5

GPU block-LU update

5.1 Introduction

Consider the standard form of the linear optimization problem

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & l_j \leq x_j \leq u_j \quad j = 1, 2, \ldots n \end{aligned} \tag{5.1}$$

where $A \in \mathbb{R}^{m \times n}$ with $m < n$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$. We assume that A has full rank, so there exists at least one set of m columns that forms a non-singular basis matrix $B \in \mathbb{R}^{m \times m}$ and the remaining columns form a matrix $N \in \mathbb{R}^{m \times n-m}$. The columns in B are denoted as the basic variables, and the remaining variables as nonbasic. We express the linear problem (5.1) as

$$\begin{aligned} \min \quad & c_B^T x_B + c_N^T x_N \\ \text{s.t.} \quad & Bx_B + Nx_N = b \\ & l_j \leq x_j \leq u_j \quad j = 1, 2, \ldots n \end{aligned} \tag{5.2}$$

where c_B, x_B and c_N, x_N correspond to the cost vectors and variables associated with the basic and nonbasic variables, respectively. Given B, a solution to the linear problem (5.2) is found by fixing the nonbasic variables to their lower or upper bound and solving $Bx_B = b - Nx_N$ for x_B. If all of the basic variables satisfy bound constraints, then the solution is a basic feasible solution. The set of all feasible solutions forms an n-dimensional polytope.

If an optimal solution to the linear program exists, at least one optimal solution point will occur on one of the vertices of this polytope. The simplex algorithm searches through these extreme vertices until obtaining an optimal solution, or determining that the problem is unbounded. Since its inception 70 years ago by George Dantzig [49], several variants of the simplex algorithm have been proposed [50, 109, 39, 134].

Over the last decade, graphics processing units (GPUs) have accelerated variants of the simplex algorithm [161, 25, 107, 139, 116, 84, 38]. GPUs are many-core processors capable of performing thousands of operations simultaneously, resulting in faster algorithms. GPUs have improved the simplex algorithm when applied to dense LP problems. Speedups have typically arisen from parallelizing matrix-matrix and matrix-vector multiplication in a tableau simplex implementation. Instead, we develop a GPU block-LU update that uses the Schur-complement. Our algorithm is developed to solve any linear programming problem. We improve algorithmic performance by replacing sparse operations with dense operations on the Schur-complement matrix that are amenable to GPU parallelization. The block-LU update best accelerates LP problems whose performance is limited by solving linear systems and LU update operations. We demonstrate the effectiveness of the block-LU update approach on problems of this type by experimenting with quantile regression problems. While linear regression is obtained through minimizing the sum of squared residuals, quantile regression is the solution to minimizing the sum of absolute residuals [103]. Quantile regression is a robust estimator that outperforms linear regression when the noise is heteroskedastic, and is more robust when generating models from data with outliers [102]. A quantile regression model is obtained from solving the following linear programming problem:

$$\min_{(\beta,u,v)\in\mathbb{R}^r\times\mathbb{R}^{2s}_+}\{\tau 1_r^T u + (1-\tau)1_r^T v | X\beta + u - v = b\}, \tag{5.3}$$

where r is the number of observations, s is the number of design variables, $\tau \in \{0, 1\}$ is

the desired quantile, $X \in \mathbb{R}^{r \times s}$ is a design matrix used for generating a regression model, $b \in \mathbb{R}^r$ is the observed output response, and $1_r \in \mathbb{R}^r$ where all values are 1. The regression coefficients are stored in β, while u and v are non-negative slack variables. The constraint matrix of linear problem (5.3), which we will denote as $A \in \mathbb{R}^{m \times n}$, where $m = r$ and $n = 2r + s$.

The primary contributions of this paper are:

1. We develop a GPU parallel block-LU update that uses the Schur-complement. We perform sparse matrix operations on the basis matrix, and dense linear algebra on the dense Schur-complement matrix.
2. We show a speedup of up to 4.03x over a CPU linear programming solver HSL LA04 for quantile regression problems. Speedups are a direct result from performing basis updates and solving systems of linear equations in parallel on the GPU.
3. We show that the GPU block-LU approach is insensitive to numerical instability that causes LA04 to prematurely return a suboptimal solution or incorrectly declare that the problem is unbounded.

In Section 5.2, we provide an algorithmic overview of the primal simplex algorithm, and review literature related to parallel implementations of the simplex algorithm and the block-LU update. In Section 5.3, we propose and outline our hybrid CPU-GPU parallel block-LU update algorithm. In Section 5.4, we present computational results between our algorithm and a primal simplex algorithm, HSL LA04, on quantile regression problems. We provide concluding remarks in Section 5.5.

5.2 Background

5.2.1 Primal simplex algorithm

We present Goldfarb and Reid's [72] steepest-edge variant.

1. Initialization: Obtain a basic feasible solution B.
2. Factorization: Factorize $B = LU$, where L is a lower triangular matrix and U is an upper triangular matrix.
3. Pricing: Solve the linear system $B^T\pi = c_B$, where $\pi \in \mathbb{R}^m$ denotes the prices of the basic variables.
4. Reduced costs: Select a candidate variable to enter the basis by calculating reduced costs for all nonbasic variables. Compute $d_j = c_j - \pi^T a_j$ for all $j \in N$, where a_j represents column j from N. Using steepest edge weighting, select the variable that minimizes the normalized reduced costs. If no variables have a positive steepest edge weight, the current solution is optimal.
5. Select an improving direction: Solve $By = a_{se}$, where a_{se} is the column of N selected from steepest edge weighting.
6. Determine the exiting variable: Use the ratio test [122] to identify which variable exits the basis when moving along the improving direction. If no variable exits the basis, the problem is unbounded.
7. Update basis: When a column in the basis is replaced by another column, the LU factors are updated. After updating, the algorithm returns to the pricing step, unless numerical instability is detected, in which case the algorithm returns to the factorization step. The algorithm may also be refactored if it is expected to improve computational efficiency.

Modern implementations of the simplex algorithm require efficient methods for solving

linear systems with B. Instead of inverting B after changing one column, update techniques reduce the number of floating point operations to solve a linear system with B in the next iteration. Dantzig and Orchard-Hays [51] proposed to represent the inverse of the basis as a product of elementary matrices. An updated version of the basis was computed from the product of the previous inverse representation and elementary matrices. Markowitz [118] proposed to use LU decomposition to decompose B, and perform updates on the LU factors in each iteration using the product form of the inverse.

Bartels and Golub [18] proposed an LU update with row pivoting to mititgate round-off errors. By updating the LU factors, the Bartels-Golub update avoids calculating the inverse of the basis, and only requires triangular solves with upper and lower triangular matrices. The Bartels-Golub update performs a rank-one update on the leaving column U, pivoting the spike column to the last column, and then using a series of elementary transformations to restore an upper triangular matrix. The Bartels-Golub update was a stability improvement compared to previous basis update methods. However, the update may produce fill-in.

Several variants were proposed to improve upon deficiencies of the Bartels-Golub update. Saunders [154] observed that the spike is unlikely to extend to the last row. Instead of permuting the spike to the last column, Saunders permutes the spike such that the last nonzero value is a diagonal element. Reid [146] implemented the Bartels-Golub update and improved the implementation by applying permutations to reduce fill-in. Another method to reduce fill-in was the Forrest-Tomlin update [65]. While the Bartels-Golub update pivots only the spike column to the last column, the Forrest-Tomlin update pivots both the rows and columns. After row and column operations, the pivoted upper triangular matrix will have no increase in the number of nonzero elements. The Forrest-Tomlin update loses the stability guarantee of the Bartels-Golub update, and instead a check needs to be performed

after the update. Fortunately, the update preserves the sparsity of the basis making this update computationally efficient when it is stable.

Another class of updates are the Schur-complement updates. These updates were originally proposed by Bisschop and Meeraus [26], who observed that the basis could be represented as an augmented basis matrix. A sparse factorization occurs on the original basis matrix B_0, and dense matrix operations are performed on a smaller Schur-complement matrix, $C_k \in \mathbb{R}^{p \times p}$, where k is the number of iterations since factorizing B_0, p is the number of rows and columns in C_k, and $p \leq k$. The system $By = a_{se}$ is then solved with the augmented system of equations

$$\begin{bmatrix} B_0 & V_k \\ E_k^T & \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{se} \\ 0 \end{bmatrix} \tag{5.4}$$

to determine an improving direction in each iteration. Here, $B_0 \in \mathbb{R}^{m \times n}$, $V_k \in \mathbb{R}^{m \times p}$, and column v_i replaces a column in B_0. The matrix $E_k \in \mathbb{R}^{m \times p}$ is a set of unit vectors e_j, where j is the column of the basis that has been replaced by vector v_i. Observing that $E_k^T y_1 = 0$, particular elements from y_1 are set to zero, and elements from y_2 are combined with y_1 to obtain the solution $y \in \mathbb{R}^m$. Similarly, the system of equations $B^T \pi = c_B$ is solved by

$$\begin{bmatrix} B_0^T & E_k \\ V_k^T & \end{bmatrix} \begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \tag{5.5}$$

where c_1 and c_2 are a permutation of $[c_b \ 0]^T$ related to the variables of the augmented system that are in the basis and π is equivalent to π_1. In this paper, we utilize the block-LU update proposed by Gill et al. [70]. For a more exhaustive list of basis update methods, we refer the interested reader to [60].

5.2.2 Block-LU update

The block-LU update involves decomposing the system of equations into two augmented matrices:

$$\begin{bmatrix} B_0^T & E_k \\ V_k^T & \end{bmatrix} = \begin{bmatrix} L_0 & \\ Z_k^T & C_k \end{bmatrix} \begin{bmatrix} U_0 & Y_k \\ & I \end{bmatrix} \tag{5.6}$$

where L_0 and U_0 are obtained from LU factorization of B_0. From (5.6), $L_0 Y_k = E_k$, $Z_k^T U_0 = V_k^T$, and $Ck = -Z_k^T Y_k$, where I is the identity matrix of size $p \times p$, $Z_k \in \mathbb{R}^{m \times p}$, and $Y_k \in \mathbb{R}^{m \times p}$. The system $By = a_{se}$ is then solved by solving three equations:

$$L_0 w = a_{se} \tag{5.7}$$

$$C_k y_2 = -Z_k^T w \tag{5.8}$$

$$U_0 y_1 = w - Y_k y_2. \tag{5.9}$$

The system $B^T \pi = c_B$ is obtained from

$$U_0^T w = c_1 \tag{5.10}$$

$$C_k^T \pi_2 = c_2 - Y_k^T w \tag{5.11}$$

$$L_0^T \pi_1 = w - Z_k \pi_2. \tag{5.12}$$

The matrices Y_k, Z_k, and C_k are stored and updated after every iteration. The matrices are updated depending on the variables that enter and exit the basis [61, 74]. If the entering variable is from N_0 and the leaving variable is from B_0, then the three matrices increase in size to store information about the incoming variable. If the entering variable is from B_0 and the leaving variable is from N_0, then the matrices decrease in size to represent the current basis. If the entering variable is from B_0 and the leaving variable is from B_0, then the matrices remain the same size, but Y_k and C_k are updated. If the entering variable is

from N_0 and the leaving variable is from N_0, then the matrices remain the same size, but Z_k and C_k are updated. From the above, the value of p is always less than or equal to k and the size of the augmented matrix is equal to $(m+p) \times (m+p)$.

From (5.8) and (5.11), it should be noted that a system of linear equations needs to be solved with the Schur-complement matrix in every iteration. Instead of explicitly updating C_k, the LU factors of the Schur-complement can be directly updated [74]. In the first case when a new column and row are added to the Schur-complement matrix, the augmented Schur-complement matrix can be expressed as:

$$C_{k+1} = \begin{bmatrix} C_k & w \\ v^T & \sigma \end{bmatrix} \tag{5.13}$$

where $v = -Y_k' Z_i$ and $w = -Z_k' Y_i$ are a new row and column added to C_k in iteration i [74]. Let $C_k = P_k L_k U_k$ be the LU factorization with partial pivoting of C_k. Then, it is possible to factor the augmented system such that

$$C_{k+1} = \begin{bmatrix} C_k & w \\ v^T & \sigma \end{bmatrix} = \begin{bmatrix} P_k & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} L_k & 0 \\ v^T U_k^{-1} & 1 \end{bmatrix} \begin{bmatrix} U_k & L_k^{-1} P_k^{-1} w \\ 0 & \rho \end{bmatrix} \tag{5.14}$$

where $\rho = \sigma - v^T U_k^{-1} L_k^{-1} P_k^{-1} w$.

There are several reasons why the block-LU update should be considered for the simplex algorithm. Instead of operating on the entire LP in each iteration, most of the computations and updates only occur on the much smaller Schur-complement matrix. Several authors have also commented that the stability of the block-LU update is comparable to that of the Bartels-Golub update, under the condition that the original basis matrix is well-conditioned [70, 62].

Finally, the block-LU update can be accelerated with parallel computing. From previous experimentation, the sparsity of matrices Y_k and Z_k has been observed on average to be 29% for problems in the Netlib [32] test library [62]. Given the density and size of these

matrices, Y_k, Z_k, and C_k are stored and operated on with dense matrix operations that are amenable to parallel computing. We propose to accelerate the block-LU update with GPU computing.

5.2.3 Parallel simplex algorithm

Algorithmic developments and advances in hardware have resulted in tremendous speedups in solving linear programming problems. Both commercial and research linear programming solvers have utilized parallel computing to solve large linear programming (LP) problems. Bixby and Martin [27] profiled the dual revised simplex algorithm and discovered several operations that benefited from parallel processing: pricing, the ratio test, and pivot selection. Huangfu and Hall [89] incorporated a parallel revised dual simplex algorithm into the FICO Xpress solver that uses the dual steepest edge method. When the inverse of the basis is sparse, Shu [156] observed a 17x speedup for some Netlib test problems by parallelizing the sparse linear algebra operations on the basis matrix. The steepest-edge pivot selection rule when parallelized on a CPU achieved a speedup of up to 1000x for dense LP problems compared to a sequential steepest-edge pivoting rule [165]. While traditionally the operations in one simplex iteration are dependent on the result of a previous iteration, the authors of [80] considered overlapping iterations. By using potentially old reduced cost information, they speedup the simplex by 2.5x to 4.8x for medium sized Netlib linear programming problems. Solving several primal subproblems simultaneously, the authors of [101] demonstrate that a parallel algorithm can solve LPs with millions of columns.

In the last decade, researchers have accelerated variants of the simplex algorithm with GPUs that have primarily focused on solving dense problems. Comparing a GPU and CPU implementation in [161], the authors showed that for dense problems a GPU implementation was slower than a sequential CPU implementation for problems with less than 1600

variables and constraints, but achieved a speedup of 2.5x for problems with 2000 variables and constraints. Using dedicated graphic shaders, the authors of [25] demonstrated a 16x speedup over GLPK using a custom GPU steepest-edge simplex algorithm on randomly generated LP problems with 50% density. Ploskas and Samaras [138] examined the impact of parallelizing pivoting rules for the simplex algorithm. They discovered that a GPU parallel steepest-edge rule led to a 16x speedup for randomly generated dense LP problems. Ploskas and Samaras [139] also demonstrated that GPU approaches speedup both dense and sparse problems. They observed modest speedups when comparing a CPU and GPU primal-dual exterior point simplex algorithm. A multi-GPU implementation of the primal simplex tableau method has been reported to be 24.5x faster than a CPU version for dense LPs with more than 20000 variables and constraints [107]. A hybrid CPU and GPU simplex tableau method was also presented in [116], where part of the tableau was offloaded to the GPU. Both processing units performed updates on the tableau and the process was repeated until an optimal solution was obtained. Dense problems with 10000 variables and constraints achieved a 30x speedup over a CPU algorithm. Speedups were also observed on several sparse problems from the Netlib test set as the number of CPU cores increased.

5.3 GPU block-LU update

In this section, we present our GPU-based CUDA Fortran implementation of the primal simplex algorithm. For the operations we do not parallelize, we use the primal simplex algorithm implemented in the HSL LA04 algorithm [75]. Designed by Goldfarb and Reid [72], the HSL LA04 is a steepest-edge simplex implementation. In our implementation, we parallelize:

1. LU factorization,
2. Selecting an improving direction,

3. Pricing,
4. Updating the basis, and
5. Revising the reduced costs.

Figure 5.1 is an overview of our algorithm, showing the operations that occur on the CPU or GPU, and when we transfer values between the processing units. After preliminary profiling of the LA04 algorithm, we determined that a hybrid CPU-GPU algorithm would be more efficient than a GPU only implementation. We parallelized the most time consuming operations, and perform the other operations on the CPU.

When solving the improving direction and pricing system of equations with the block-LU update, we develop two kernels: blockBTRAN and blockFTRAN. We implement blockBTRAN and blockFTRAN with cuSPARSE [129], cuSOLVER [128], cuBLAS [42], and CUDA Fortran. Algorithm 3 and Algorithm 4 are pseudocode for our implementations. Algorithm 3 accepts the column entering the basis as an input. Since A is stored on the GPU, we gather the nonzeros of the incoming column from GPU memory without transferring data between the processing units. We then solve three systems of linear equations for y and retain the vector a_i for use in the Schur-complement update. In Algorithm 4, we load cost coefficients from GPU memory. As U_0 and c_1 remain the same until the basis is refactored, after calculating w in the first iteration, Step 4 in the algorithm is replaced by loading w from GPU memory. After solving for the reduced costs, π is set equal to π_1 and π_2 is discarded.

Parallelism in both blockBTRAN and blockFTRAN arises during (a) triangular solves, (b) matrix-vector operations, and (c) operations with the Schur-complement matrix. Triangular solves are inherently sequential operations, where the values of some variables are dependent on other values. Yet, the authors of [124] were able to develop a parallel method for solving a triangular system in two steps. First, analyze the structure of the matrix

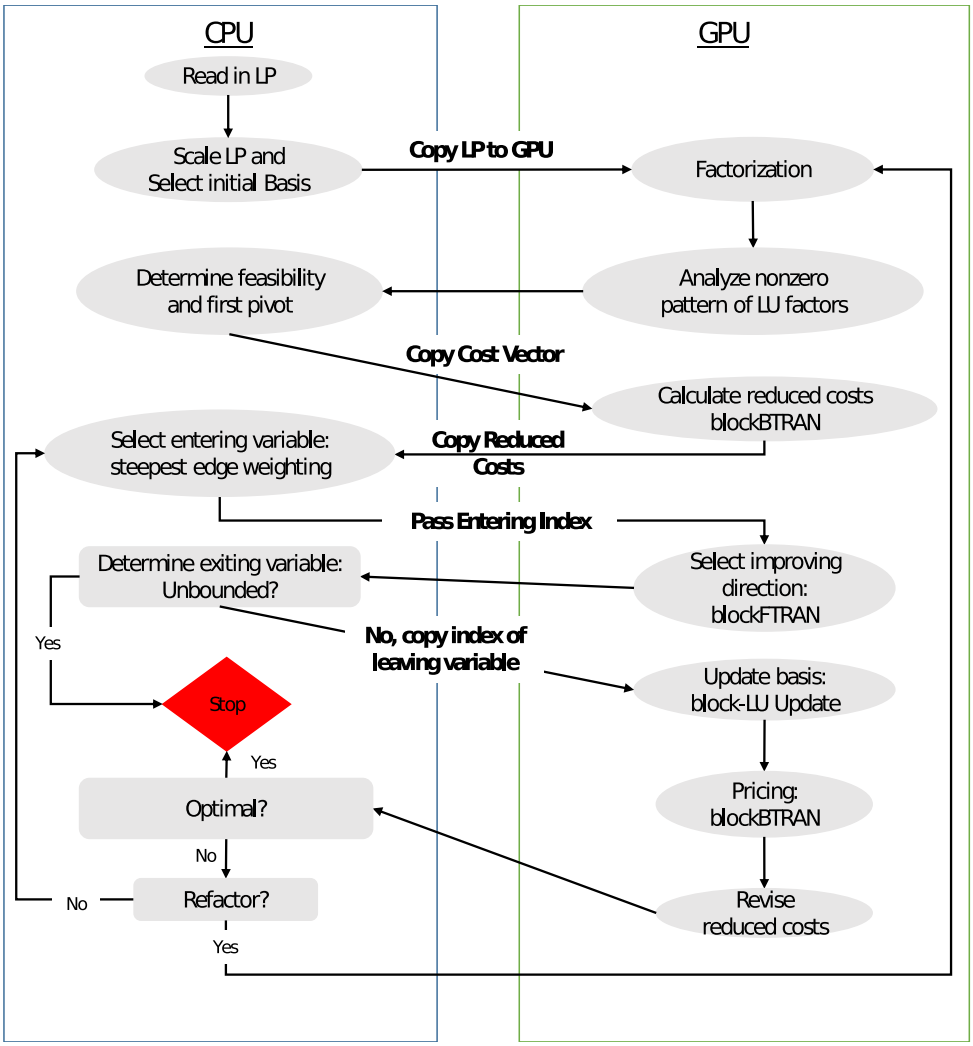


Figure 5.1: Overview of the hybrid block-LU update implementation. Operations in the left box take place on the CPU, while operations in the right box take place on the GPU. Lines that cross between the boxes represent data transferred between processing units.

Algorithm 3 GPU implementation of blockFTRAN for solving $By = a_{se}$. This function uses the block-LU update to solve for vector y. In Step 4 and Step 8 we perform a triangular solve using L_0 and U_0. These matrices are produced from LU factorization of the basis matrix, and unchanged until refactorization.

1: Given an initial factorization L_0 and U_0, p the number of rows replaced in B_0, C_k, Y_k, Z_k, and column i selected from steepest edge weighting:
2: **procedure** BLOCKFTRAN(i, p)
3: Gather the sparse vector a_i from GPU storage.
4: Using L_0, perform a parallel triangular solve for w with Equation (5.7).
5: **if** $p \geq 0$ **then**
6: Perform a triangular solve for y_2 with Equation (5.8).
7: **end if**
8: Solve for y_1 with Equation (5.9).
9: Copy y_1 and y_2 from the GPU to the CPU.
10: Using the variables currently in the basis, replace values from y_1 with the corresponding values in y_2.
11: Retain vector a_i in GPU memory for use in the next LU update.
12: **end procedure**

Algorithm 4 GPU implementation of blockBTRAN for solving $B^T\pi = c_B$. The block-LU update is used to solve for the reduced costs pi. Step 4 of this algorithm only needs to be performed once for every matrix factorization. As c_1 are the basis matrix cost coefficients, and U_0 is only updated upon factorization, w does not change. In practice, we store e in GPU memory and access it in subsequent iterations.

1: Given an initial factorization L_0 and U_0, p the number of rows replaced in B_0, C_k, Y_k, Z_k, and the cost for each value in the basis c_1 and c_2:
2: **procedure** BLOCKBTRAN(c_1, c_2, p)
3: Load c_1 and c_2 from GPU memory.
4: Using U_0^T solve for w with Equation (5.10).
5: **if** $p \geq 0$ **then**
6: Solve for π_2 with Equation (5.11).
7: **end if**
8: Solve for π_1 with Equation (5.12).
9: Copy π_1 from the GPU to the CPU and set $\pi = \pi_1$.
10: **end procedure**

and group independent rows into separate levels that can be solved simultaneously. Then, perform the solve operation exploiting as much parallelism as possible. We utilize the cusparseDcsrsv2 routine that implements this parallel triangular solve [129]. Given that the block-LU update relies on factoring an initial basis and then performing hundreds of solves with the same basis factorization, the cost of the analysis phase is amortized over hundreds of iterations. When we factor a new basis matrix, we also perform the requisite analysis phases. Second, in both blockBTRAN and blockFTRAN, dense matrix-vector multiplication occurs during (5.9) and (5.12). By explicitly storing Y_k and Z_k as dense matrices, we parallelize these operations with dense GPU matrix-vector multiplication kernels. Finally, as we store our Schur-complement matrix as a dense matrix, we use the cusolverDnDgetrs routine to solve (5.8) and (5.11).

As previously mentioned, four different updates may occur in each iteration using the Schur-complement update. In each iteration, we determine which update is required and then update C_k, Y_k, or Z_k. Updates are performed using sparse triangular solve operations. After updating Y_k and Z_k, we then update C_k depending on which columns were swapped in the current iteration. If a column from B_0 is replaced by a column from N_0, causing the Schur-complement to grow in size, we perform an LU update on the Schur-complement as done in [74] and as described in Section 5.2.2. Otherwise, $C_k = -Z_k' Y_k$, and we perform a dense LU factorization on the new matrix C_k. As the size of C_k is small compared to the size of general linear programming problems, this method is appropriate as long as the maximum dimension of C_k is limited. For quantile regression problems, we found that limiting the size of the Schur-complement matrix to at most 1024×1024 was numerically stable, and produced an efficient algorithm. Updating the reduced costs requires updating all nonbasic variables. Fortunately, this involves a set of single-instruction multiple data operations that we parallelized with a GPU kernel.

Inspired by work from Bisschop and Meeraus [26], Eldersveld and Rinard [61] developed a parallel block-LU update that uses the Schur-complement on a vector machine. They parallelized triangular solves on a multi-core computer and demonstrated a moderate speedup with vectorization. Our hybrid simplex algorithm is a modification of the block-LU update implemented by Eldersveld and Saunders [62]. The novelty of our implementation is that (a) we explicitly store Y_k and Z_k as dense matrices, (b) we perform dense linear algebra when operating on the Schur-complement matrix, and (c) we apply an LU updating strategy to the Schur-complement matrix C_k.

5.4 Computational results

We conducted experiments on a machine running CentOS7, with an Intel Xeon E5-1630 at 3.7 GHz and 8 GB of RAM, with a NVIDIA Tesla K40 GPU with 15 streaming multiprocessors, and 12 GB of RAM. Algorithms were compiled with the PGI-19.10 pgfortran compiler that linked to the CUDA 10.1 library. All files were compiled with the -fast, -O3, and -Kieee flags. We experimented with randomly generated dense quantile regression problems. Similar to the work of [111], we generated random design matrices with $r = \{2000, 2500, 3000\}$ $s = 1500$, and we experimented with $\tau = \{0.1, 0.3, 0.5, 0.7\}$. The values in the design matrix were generated randomly from $\{0, 1\}$, and $b = X\beta + 1/(s+1)\mathcal{N}(0, 1)$, and β is also randomly drawn from $\{0, 1\}$. For each combination of r, s, τ, we experiment with three randomly generated problems. The values reported in Table 5.1 are the average from three experiments.

We present our computational results in Table 5.1. The first three columns are properties of the design matrix, and the last three columns are the execution time of the CPU simplex algorithm, the GPU simplex algorithm, and the hybrid speedup. As two slack variables are included for every observation, the dimensions of the linear programming problem are r by

Table 5.1: HSL LA04 and hybrid CPU-GPU primal simplex comparison

r	s	τ	Hybrid CPU-GPU time (s)	HSL LA04 time (s)	Hybrid speedup
2000	1500	0.1	237.78	244.46	1.04
2500	1500	0.1	556.85	626.06	1.15
3000	1500	0.1	844.92	3402.05	4.03
2000	1500	0.3	372.45	581.96	1.55
2500	1500	0.3	868.45	1281.12	1.50
3000	1500	0.3	1478.52	2652.52	1.82
2000	1500	0.5	406.00	534.33	1.29
2500	1500	0.5	913.84	1622.15	1.79
3000	1500	0.5	1414.82	3162.58	2.30
2000	1500	0.7	354.52	393.31	1.11
2500	1500	0.7	847.69	1212.65	1.42
3000	1500	0.7	1436.75	2801.03	2.03

$2r + s$. The speedup was calculated as

$$CPUTime(s)/HybridTime(s) = Speedup. \tag{5.15}$$

Our parallel CPU-GPU block-LU update outperforms the LA04 simplex algorithm. The largest speedups in our algorithm were a result of operating on the dense Schur-complement instead of the entire LP problem. As we are operating on dense matrices, our algorithm benefits from parallelizing both the matrix-vector multiplication and solving systems of equations with the Schur-complement matrix. Across different values of τ, we observed a geometric mean speedup of 1.23x for problems with 2000 rows, 1.45x for problems with 2500 rows, and 2.42x for problems with 3000 rows. Given that the speedup tends to increase

as problem size increases, our results suggest that the speedups may increase if we were to experiment on larger problems. In the best case, we demonstrate a 4.03x speedup for problems with $r = 3000$ and $\tau = 0.1$.

With our selected maximum size of the Schur-complement matrix, the block-LU update performed thousands of updates before refactorizing. The time spent analyzing the underlying structure of L_0 and U_0 was less than the time saved by reducing the time spent performing thousands of triangular solves. We also observed that the time spent on blockFTRAN and blockBTRAN was less than the time spent performing triangular solves in the LA04 algorithm. Updating the reduced costs had a small effect on performance, but tended to reduce execution time by a marginal amount.

In addition to performance differences from using the block-LU update, we also observed that the two algorithms took different solution paths. While both algorithms will refactor if numerical instability is detected, refactorization for our GPU algorithm occurs when p equals the maximum size of the Schur-complement matrix, and the LA04 algorithm is refactored when it detects a performance decrease. We observed that, for quantile regression problems, our parallel block-LU algorithm needed less iterations than the LA04 algorithm. This is related to the stability of the block-LU update. The block-LU update is as stable as the Bartels-Golub update assuming that the initial basis is well-conditioned [62]. In our experimentation, we never encountered numerical instability with the block-LU update. Additionally, the block-LU update always terminated with the optimal solution for all of the problems that we considered. For quantile regression problems with 2500 or 3000 rows, the HSL LA04 algorithm sometimes returned a suboptimal solution or incorrectly declared the problem as unbounded.

We observed that, by adjusting the condition for when LA04 refactors the basis, we were able to improve the likelihood that the algorithm terminated at an optimal solution.

The LA04 algorithm defaults to refactoring whenever the execution time in the last 10 iterations is 5% slower than the previous 10 iterations. When we lowered the threshold before refactoring the basis, we found that the LA04 algorithm experienced less numerical instability. For example, with the default setting, after solving the 3000 row, 1500 column, and $\tau = 0.7$ problem 10 times, LA04 returned the optimal solution three times, and seven times out of ten when the algorithm refactored after a 2.5% slowdown.

5.5 Conclusions

This paper investigates parallelizing the block-LU update in a primal simplex algorithm. Despite the potential performance benefit of graphics processing units, no work has been done to utilize GPUs for performing the block-LU update in the simplex algorithm. We develop a hybrid CPU-GPU primal simplex algorithm that utilizes the GPU to perform the block-LU update. We parallelize sparse LU factorization, the block-LU update, revising reduced costs, and performing BTRAN and FTRAN operations on the GPU.

We show the effectiveness of our approach on quantile regression problems formulated as linear programming problems. We obtain up to a 4.03x speedup over a CPU linear programming solver HSL LA04. We observe that for problems with 2000-4000 observations and 1500 features our algorithm has a geometric mean performance that is 1.63x faster than LA04. We demonstrate that the block-LU update does not suffer from numerical instability that LA04 can experience for quantile regression problems with 2500 or 3000 rows.

Chapter 6

Conclusion

6.1 Summary of this Thesis

In this section, we summarize important findings and major accomplishments from each chapter.

6.1.1 Chapter 2: LLSP Parameter Tuning

In Chapter 2, we improve the performance of a GPU parallel linear least squares solver for tall and skinny matrices that are common in model building. With the goal of accelerating the generation of empirical models, we compare linear least squares solvers and identify that the MAGMA [93] linear least squares routine, a hybrid CPU-GPU approach, outperformed a CPU only approach [9] and a GPU only approach [128]. However, in the case of overdetermined problems that routinely arise when generating empirical models, we noticed a decrease in the performance of the MAGMA linear least squares solver.

We conduct an extensive computational comparison of different derivative-free optimization and simulation optimization algorithms. By tuning the block size parameter, we improve the performance of MAGMA on tall and skinny matrices by a factor of 1.8x. Five derivative-free optimization solvers identify optimal tuning parameters for a variety of tall and skinny matrices with an order of magnitude less simulations than complete enumeration. The derivative-free optimization solvers also identify higher quality solutions faster than OpenTuner [12].

6.1.2 Chapter 3: HybridTuner

In Chapter 3, we return to the problem of algorithmic parameter tuning. We review the field of autotuning with a focus on techniques that utilize derivative-free optimization search strategies. Autotuning is essential for parameter tuning to allow for performance portability. To cover different potential use cases, algorithms may end up with hundreds of tunable parameters that need to be adjusted in the context of a specific problem. GPU architectures are especially challenging to tune as new architectures are released every other year, and optimal parameters on one system may not even compile yet alone be optimal on another system. Then as the number of tuning parameters increases, the computational cost of identifying optimal tuning parameters increases almost exponentially.

We focus on two tuning applications: a GPU dense matrix-matrix multiplication kernel and tuning the GCC compiler. The efficiency of dense matrix multiplication directly impacts the performance of several algorithms ranging from training neural networks to linear and nonlinear optimization problems. We tune a customized GPU matrix-matrix multiplication algorithm with 17 tuning parameters. To demonstrate the performance of autotuners on a more complex problem, we tune the GCC compiler. Tuning GCC compiler options beyond the default -O2 or -O3 flags is a daunting task with hundreds of parameters [13, 14, 12, 33, 164]. This is exacerbated when considering that there is no explicit algebraic function to relate tunable parameters to an output performance metric, and that these models, if they did exist, would be dependent on problem size and computer architecture.

To address these challenges, we propose to perform autotuning with novel hybrid derivative-free optimization strategies. We review the literature and discover that several autotuners utilize local derivative-free optimization search strategies [12, 163] such as a Nelder-Mead [125] simplex search. Inspired by results in Chapter 2, we investigate the benefit of combining local and global derivative-free optimization search strategies. In particular, we

initialize a search strategy with the result from the DIRECT algorithm [96]. We propose two hybrid derivative-free optimization initialization strategies: Hybrid DFO and Bandit DFO. Hybrid DFO, uses the DIviding RECTangles (DIRECT) [96] algorithm to generate a starting point for a single local derivative-free optimization strategy. The second, Bandit DFO, uses the DIRECT algorithm to generate a starting point and then utilizes the multi-armed bandit function with area under the curve credit assignment problem [133] to identify near optimal solutions for all of the problems that we consider.

We improve the performance of dense matrix-matrix multiplication by a factor of 1.4x compared to optimal parameters identified by OpenTuner, ActiveHarmony, and a popular strategy in neural network tuning, Bayesian Optimization. Bandit DFO speeds up algorithms compiled by GCC, identifying the best observable parameters with 5% of the iterations required by other methods. Finally, we make our code HybridTuner publicly available to facilitate the development of future autotuning methodologies.

6.1.3 Chapter 4: Backward Stepwise Elimination

In Chapter 4, we extend our work in Chapter 2 from solving a single linear model to the subset selection problem. Instead of solving a regression problem, we consider parallel methods for selecting a subset of features to use in a linear model. Sparse models generated in this way prevent over-fitting, can be integrated into an optimization model, and solved with generic optimization solvers. We utilize the backward stepwise elimination algorithm as it is amenable to parallel computing and demonstrate that it comes with an approximation guarantee.

Using the supermodularity ratio, we develop an approximation guarantee for backward stepwise elimination. Despite the fact that the approximation guarantee deteriorates when a function is far away from being supermodular, we are able to provide some confidence

in backward stepwise elimination in certain cases that had not existed previously. We also present a GPU parallel batched backward stepwise elimination algorithm, evaluating hundreds of potential models simultaneously to select the best one. Computational results suggest that the GPU algorithm is 5x faster than a sequential CPU approach. Finally, we systematically compare the accuracy of backward stepwise elimination against other subset selection approaches. For certain classes of problems, backward stepwise elimination selects models that are sparser and more accurate than the lasso or forward selection. This work expands upon recent contributions in the statistics community [83] to compare the accuracy of subset selection techniques and suggest that different methods are applicable under different scenarios.

6.1.4 Chapter 5: GPU block-LU update

In Chapter 5, we solve linear programming problems with the primal simplex algorithm. In particular, quantile regression problems. Quantile regression can be used for model building when the conditions for linear regression are not met, such as when the data is heteroskedastic, nonlinear, or has outliers. For example, quantile regression has been used to suggest public policies to reduce the number of infants born with a low birthweight as low birthweights have been related to health complications [103].

To solve quantile regression problems, we develop a GPU parallel block-LU update that parallelizes the basis update and triangular solves steps in the simplex algorithm. We review the primal simplex algorithm, and describe previous approaches at accelerating it with CPU, GPU, and hybrid approaches. We then describe our parallel implementation, where we utilize dense linear algebra operations that are amenable to GPU computing to perform updates with the Schur-complement matrix. We demonstrate the efficiency of our approach with GPU speedups obtained from solving tall and skinny quantile regression

problems. We obtain up to a 4.03x speedup over the HSL LA04 simplex algorithm, and a geometric average of 1.63x over all of the problems that we consider.

6.2 Research Contributions

The major contributions to this thesis are:

1. We performed a computational comparison of derivative-free optimization and simulation optimization algorithms, comparing different classes of derivative-free optimization on an integer black-box optimization problem. We added to a literature that is still in its infancy and provided comparisons to help users determine what derivative-free optimization or simulation optimization strategy is best suited for their particular needs.

2. We accelerated MAGMA's QR solver by a factor of 1.8x for large tall and skinny matrices compared to the performance of the MAGMA algorithm with default parameters, improving its efficiency and allowing for its incorporation into a subset selection framework.

3. We identified a set of five derivative-free optimization solvers that were capable of determining optimal GPU parameters with an order of magnitude less simulations than exhaustive enumeration, reducing the computational cost to obtain an optimal algorithm regardless of system architecture.

4. For the problem of tuning QR factorization, we showed that the best DFO solvers identify high quality solutions faster than other autotuners such as OpenTuner.

5. We developed a novel algorithmic framework to combine local and global derivative-free optimization approaches for algorithmic parameter tuning, identifying the best

or near-optimal parameters for all problems considered in this work.

6. We improved the performance of dense matrix-matrix multiplication by a factor of 1.4x compared to the performance with optimal parameters identified by OpenTuner [12], ActiveHarmony [163], or Bayesian Optimization [17] after at most 1000 simulations.

7. We optimized the performance of algorithms compiled by GCC with fewer than 5% of the simulations required by other autotuners, greatly reducing the cost for autotuning and improving the portability of algorithms.

8. We shared HybridTuner with the community to help with the development and use of hybrid autotuning algorithms. Hybrid DFO and Bandit DFO can be used with any derivative-free optimization solver allowing for the creation of more sophisticated techniques.

9. We obtained an approximation guarantee for the accuracy of backward stepwise elimination using the supermodularity ratio, allowing us to understand the type of modeling conditions conducive to the generation of sparse and accurate models with backward stepwise elimination.

10. We proposed a GPU parallel batched backward stepwise elimination algorithm that was a factor of 5x faster than a CPU implementation.

11. We compared the performance of several state-of-the-art subset selection methodologies and observed that backward stepwise elimination generated models that were simpler and have less out-of-sample test error than the lasso or forward selection.

12. We developed a GPU parallel block-LU update that uses the Schur-complement. We perform sparse matrix operations on the basis matrix, and dense linear algebra on the dense Schur-complement matrix.

13. We showed a speedup of up to 4.03x over a CPU linear programming solver HSL LA04 for quantile regression problems. Speedups were a direct result from performing basis updates and solving systems of linear equations in parallel on the GPU.

14. We showed that the GPU block-LU approach is insensitive to numerical instability that causes LA04 to prematurely return a suboptimal solution or incorrectly declare that the problem is unbounded.

6.3 Papers produced by the author of this dissertation

1. B. Sauk, N. Ploskas, and N. V. Sahinidis. GPU paramter tuning for tall and skinny dense linear least squares problems. *Optimization Methods and Software*, 35:638–660, 2020

2. B. Sauk and N. V. Sahinidis. Backward stepwise elimination: Approximation guarantee, a batched GPU algorithm, and empirical investigation. *Journal of the American Statistical Association*, Submitted, 2020

3. B. Sauk and N. V. Sahinidis. HybridTuner: Tuning with hybrid derivative-free optimization initialization strategies. *ACM Transactions on Architecture and Code Optimization (TACO)*, Submitted, 2020

4. B. Sauk, N. Ploskas, and N. V. Sahinidis. A GPU parallel block-LU update. In preparation, 2020

5. O. Sarwar, B. Sauk, and N. V. Sahinidis. A Discussion on Practical Considerations with Sparse Regression Methodologies. *Statistical Science*, Submitted, 2020

6.4 Future Work

In this section, we suggest future research directions.

6.4.1 Integrate batched backward stepwise elimination into ALAMO

Automated learning of algebraic models for optimization (ALAMO) is a surrogate model building framework that generates models using a mixed integer programming formulation [44]. The proposed backward stepwise elimination algorithm could be incorporated into the ALAMO framework as another option for surrogate modeling. For large problems, mixed integer optimization may prove computationally expensive when generating a surrogate model. Currently, ALAMO uses a forward stepwise selection heuristic to generate an initial model that may be proposed to users who do not require an optimal solution. In this context, and driven by results that we have observed in Chapter 4, it may prove beneficial to provide users the option to generate a model with backward stepwise elimination.

In the context of iterative model building, one of the most important features of ALAMO is its ability to perform error maximization sampling to identify future points to evaluate and improve the accuracy of a surrogate model. This capability of updating solutions with new data points could be included in the batched backward stepwise elimination algorithm.

6.4.2 Evaluate the performance of HybridTuner

One of the main contributions from Chapter 2 is the computational comparison of derivative-free optimization and simulation optimization algorithms. This comparison could be augmented by including Hybrid DFO and Bandit DFO. A comprehensive test set for derivative-free optimization problems was proposed in [147], and can serve as a performance comparison for Hybrid DFO and Bandit DFO. Despite the fact that dense matrix multiplication and GCC tuning are challenging problems, without knowing the underly-

ing model relating input parameters to execution time, it is difficult to provide conclusions about when Hybrid DFO or Bandit DFO should be recommended for general derivative-free optimization problems.

Requiring algorithmic tuning, Bandit DFO and Hybrid DFO have a set of hyperparmeters. Performance on a test library could serve to identify better default hyperparameters for both of these algorithms. It would also be interesting to compare the performance of HybridTuner against other autotuning algorithms for tuning neural networks. The accuracy and performance of neural networks is dependent on the selection of tuning parameters, and reducing the training time with efficient autotuners, may lead to improved performance.

6.4.3 Improvements to Bandit DFO

The current framework of Bandit DFO could be expanded to accommodate more global derivative-free optimization algorithms. Currently, the framework only allows for the use of derivative-free optimization algorithm that accept a starting point. Without this restriction, a solver that is selected may not be able to use any information previously obtained, and provide no guarantees of convergence, as cycling could occur. However, some derivative-free optimization solvers could theoretically be adapted to accept a starting point and could be used in this framework.

Another interesting extension would be considering more historical information with each derivative-free optimization solver. Currently, the only information that a solver receives is a starting point, and then any information it generates during its n iterations, where n is the number of iterations between solving the multi-armed bandit problem. A solver that is developed to generate a surrogate model or to incorporate all previously existing trial points in determining the next point to evaluate would also benefit from Bandit DFO. While the ability to incorporate past information would be beneficial, this may prove challenging

to implement in practice. Derivative-free optimization algorithms in general are complex to develop, and require a lot of effort to develop theoretical performance guarantees. We provide Bandit DFO as open-source code to help promote the development of advanced derivative-free optimization solvers that could make use of these tools.

6.4.4 GPU Parallel Sparse Matrix-Vector Multiplication

An important and time intensive portion of the simplex algorithm is sparse matrix-vector multiplication (SpMV). In the case of the dual simplex algorithm, Bixby and Martin observed that SpMV accounted for 18.4% of the total time spent in the simplex algorithm for 30 sparse linear programming problems [27]. SpMV is present in the pricing step, selecting an improving direction, and revising column weights in the primal simplex algorithm. In our GPU block-LU update, we utilize dense matrix-vector multiplication. However, as problems increase in size, and when the Y_k and Z_k matrices are sparse, an optimized SpMV routine would decrease the time spent on solving the pricing and improving direction systems of equations.

To make use of existing sparse algorithms [129], we store our sparse matrix in the compressed sparse column storage format. However, there exist more than 10 other storage formats that may be more efficient for SpMV for certain types of problems. A recently proposed merge-based CSR format is particularly promising [120]. The merge-based CSR was demonstrated to perform well on sparse problems regardless of its sparsity pattern. To make use of this storage format, other sparse matrix algorithms would have to be developed, such as LU factorization, and triangular solve kernels. The solve kernel is particularly challenging as the structure of the sparse matrix needs to be analyzed to overlap computations during a triangular solve.

An interesting approach to explore is simultaneously storing multiple versions of a matrix

in different storage formats. For smaller problems, the memory and time overhead for creating and storing multiple sparse matrices in different storage formats may be smaller than the potential benefit from using different formats for different operations. As GPU memory is limited, this will become prohibitively expensive for larger sparse linear programming problems of interest, but could prove beneficial for some problems.

Bibliography

[1] M. Abalenkovs, A. Abdelfattah, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. Yarkhan. Parallel programming models for dense linear algebra on heterogeneous systems. *Supercomputing Frontiers and Innovations*, 2:67–86, 2015.

[2] M. A. Abramson, C. Audet, G. Couture, J. E. Dennis, Jr., and S. Le Digabel. The NOMAD project, Current as of 19 July, 2019. `http://www.gerad.ca/nomad/`.

[3] B. M. Adams, W. J. Bohnhoff, K. R. Dalbey, J. P. Eddy, M.S. Eldred, D. M. Gay, K. Haskell, P. D. Hough, and L. P. Swiler. `DAKOTA`*, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 5.2 User's Manual*. Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2011.

[4] B. M. Adams, M. S. Ebeida, M. S. Eldred, G. Geraci, J. D. Jakeman, K. A. Maupin, J. A. Monschke, L. P. Swiler, J. A. Stephens, D. M. Vigil, T. M. Wildey, W. J. Bohnhoff, K. R. Dalbey, J. P. Eddy, R. W. Hooper, K. T. Hu, P. D. Hough, E. M. Ridgway, and A. Rushdi. `DAKOTA`*, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.5 User's Manual*. Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2016. `https://dakota.sandia.gov/`.

[5] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR factorization on a multicore node enhanced with multiple GPU accelerators.

In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 932–943, 2011.

[6] E. Agullo, J. Dongarra, R. Nath, and S. Tomov. A fully empirical autotuned dense QR factorization for multicore architectures. In *European Conference on Parallel Processing*, pages 194–205, 2011.

[7] E. Amaldi and V. Kann. On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems. *Theoretical Computer Science*, 209:237–260, 1998.

[8] S. Amaran, N. V. Sahinidis, B. Sharda, and S. J. Bury. Simulation optimization: A review of algorithms and applications. *Annals of Operations Research*, 240:351–380, 2016.

[9] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[10] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 48–58, 2011.

[11] J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation,* Association for Computing Machinery, New York, NY, pages 38–49, 2009.

[12] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelly, J. Bosboom, UM. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 303–315, 2014.

[13] A. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51:1–42, 2018.

[14] A. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano. COBAYN: Compiler autotuning framework using Bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13:1–26, 2016.

[15] C. Audet and J. E. Dennis Jr. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization*, 17:188–217, 2006.

[16] C. Audet and D. Orban. Finding optimal algorithmic parameters using derivative-free optimization. *Society for Industrial and Applied Mathematics*, 17:642–664, 2006.

[17] M. Balandat, B. Karrer, D.R. Jiang, S. Daulton, Benjamin B. Letham, A. Wilson, and E. Bakshy. BoTorch: Programmable Bayesian Optimization in PyTorch. *arXiv preprint arXiv:1910.06403*, pages 1–20, 2019.

[18] R. Bartels and G. Golub. The simplex method of linear programming using LU decomposition. *Communications of the ACM*, 12:266–268, 1969.

[19] R. R. Barton. Metamodeling: A state of the art review. *Proceedings of the 1994 Winter Simulation Conference*, pages 237–244, 1994.

[20] C. J. Bélisle, H. E. Romeijn, and R. L. Smith. Hit-and-run algorithms for generating multivariate distributions. *Mathematics of Operations Research*, 18:255–266, 1993.

[21] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[22] R. Bendel and A. Afifi. Comparison of stopping rules in forward "stepwise" regression. *Journal of the American Statistical Association*, 72:46–53, 1977.

[23] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor and R.S. Zemel and P.L. Bartlett and F. Pereira and K.Q. Weinberger (eds.)*, Proceedings of the 24th International Conference on Neural Information Processing Systems,* Curran Associates Inc., Red Hook, NY, pages 2546–2554, 2011.

[24] D. Bertsimas, A. King, and R. Mazumder. Best subset selection via a modern optimization lens. *The Annals of Statistics*, 44:813–852, 2016.

[25] J. Bieling, P. Peschlow, and P. Martini. An efficient GPU implementation of the revised simplex method. In*, IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW),* IEEE, pages 1–8, 2010.

[26] J. Bisschop and A. Meeraus. Matrix augmentation and partitioning in the updating of the basis inverse. *Mathematical Programming*, 13:241–254, 1977.

[27] R. Bixby and A. Martin. Parallelizing the dual simplex method. *INFORMS Journal on Computing*, 12:45–56, 2000.

[28] Å. Björck, H. Park, and L. Eldén. Accurate downdating of least squares solutions. *SIAM Journal Matrix Analysis and Applications*, 15:549–568, 1994.

[29] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, and G. Henry. An updated set of basic linear algebra

subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2002.

[30] A. J. Booker, J.E. Dennis Jr., P. D. Frank, D. B. Serafini, V. J. Torczon, and M. W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. *Structural Optimization*, 17:1–13, 1999.

[31] R. P. Brent. *Algorithms for Minimization without Derivatives.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[32] S. Browne, J. Dongarra, E. Grosse, and T. Rowan. Netlib mathematical software repository, Current as of 19 June, 2020. `https://www.netlib.org/lp/data/index.html`.

[33] P. Bruel, M. Gonzalez, and A. Goldman. Autotuning GPU compiler parameter using OpenTuner. In *XXII Symposium of Systems of High Performance Computing,* IEEE, Bangalore, India, pages 1–12, 2015.

[34] J. Cadzow. Least squares, modeling, and signal processing. *Digital Signal Processing*, 4:2–20, 1994.

[35] R. Carter, J. Gablonsky, A. Patrick, C. Kelley, and O. Eslinger. Algorithms for noisy problems in gas transmission pipeline optimization. *Optimization and engineering*, 2:139–157, 2001.

[36] K.-H. Chang. Stochastic Nelder-Mead simplex method–A new globally convergent direct search method for simulation optimization. *European Journal of Operational Research*, 220:684–694, 2012.

[37] K.-H. Chang, L. J. Hong, and H. Wan. Stochastic trust-region response-surface method (STRONG)—A new response-surface framework for simulation optimization. *INFORMS Journal on Computing*, 25(2):230–243, 2013.

[38] J. Charlton, S. Maddock, and P. Richmond. Two-dimensional batch linear programming on the GPU. *Journal of Parallel and Distributed Computing*, 126:152–160, 2019.

[39] A. Charnes, A. Henderson, and W. Cooper. *An introduction to linear programming.* John Wiley & Sons, 1955.

[40] R. Chen, Y. Tsai, and W. Wang. Adaptive block size for dense QR factorization in hybrid CPU–GPU systems via statistical modeling. *Parallel Computing*, 40:70–85, 2014.

[41] COIN-OR Project. Derivative Free Optimization. `http://projects.coin-or.org/Dfo`.

[42] NVIDIA Corporation. cuBLAS, Current as of 2 June, 2020. `https://docs.nvidia.com/cuda/cublas/index.html`.

[43] C. Couvreur and Y. Bresler. On the optimality of the backward greedy algorithm for the subset selection problem. *SIAM Journal on Matrix Analysis and Applications*, 21:797–808, 2000.

[44] A. Cozad, N. V. Sahinidis, and D. C. Miller. Automatic learning of algebraic models for optimization. *AIChE Journal*, 60:2211–2227, 2014.

[45] T. Csendes, L. Pál, J. O. H. Sendín, and J. R. Banga. The GLOBAL optimization method revisited. *Optimization Letters*, 2:445–454, 2008.

[46] A. L. Custódio, H. Rocha, and L. N. Vicente. Incorporating minimum Frobenius norm models in direct search. *Computational Optmization and Applications*, 46:265–278, 2010.

[47] A. L. Custódio and L. N. Vicente. Using sampling and simplex derivatives in pattern search methods. *SIAM Journal on Optimization*, 18:537–555, 2007.

[48] A. L. Custódio and L. N. Vicente. *SID-PSM: A pattern search method guided by simplex derivatives for use in derivative-free optimization*. Departamento de Matemática, Universidade de Coimbra, Coimbra, Portugal, 2008.

[49] G. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysis of production and allocation*, 13:339–347, 1951.

[50] G. Dantzig. Notes on Linear Programming—Part III: Computational Algorithm of the Revised Simplex Method. *RAND Report*, 1953.

[51] G. Dantzig and Wm. Orchard-Hays. The product form for the inverse in the simplex method. *Mathematical Tables and Other Aids to Computation*, 8:64–67, 1954.

[52] A. Das and D. Kempe. Approximate submodularity and its applications: subset selection, sparse approximation and dictionary selection. *The Journal of Machine Learning Research*, 19:74–107, 2018.

[53] A. Davidson and J. Owens. Toward techniques for auto-tuning GPU algorithms. In Jónasson K. (ed.), *Applied Parallel and Scientific Computing,* Springer, Berlin, Heidelberg, pages 110–119, 2012.

[54] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[55] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34:A206–A239, 2012.

[56] T. Dong, A. Haidar, P. Luszczek, S. Tomov, A. Abdelfattah, and J. Dongarra. MAGMA Batched: A Batched BLAS Approach for Small Matrix Factorizations and Applications on GPUs. Technical report, Technical Report, 2016.

[57] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, 1995.

[58] M.A. Efroymson. Multiple regression analysis. In A. Ralston and H.S. Wilf (eds.), *Mathematical Methods for Digital Computers,* Wiley, New York, pages 191–203, 1960.

[59] J. Elble, N. V. Sahinidis, and P. Vouzis. GPU computing with Kaczmarz's and other iterative algorithms for linear systems. *Parallel computing*, 36:215–231, 2010.

[60] J. Elble and N.V. Sahinidis. A review of the LU update in the simplex algorithm. *International Journal of Mathematics in Operational Research*, 4:366–399, 2012.

[61] S. Eldersveld and M. Rinard. A vectorization algorithm for the solution of large, sparse triangular systems of equations. Technical report, STANFORD UNIV CA SYSTEMS OPTIMIZATION LAB, 1990.

[62] S. Eldersveld and M. Saunders. A block-LU update for large-scale linear programming. *SIAM Journal on Matrix Analysis and Applications*, 13:191–201, 1992.

[63] S. S. Fan and E. Zahara. A hybrid simplex search and particle swarm optimization for unconstrained optimization. *European Journal of Operational Research*, 181:527–548, 2007.

[64] A. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60:25–64, 2010.

[65] J. Forrest and J. Tomlin. Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical programming*, 2:263–278, 1972.

[66] J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33, 2010.

[67] G. Furnival and R. Wilson. Regressions by leaps and bounds. *Technometrics*, 16:499–511, 1974.

[68] C. Gatu and E. Kontoghiorghes. Branch-and-bound algorithms for computing the best-subset regression models. *Journal of Computational and Graphical Statistics*, 15:139–156, 2006.

[69] E. George. The variable selection problem. *Journal of the American Statistical Association*, 95:1304–1308, 2000.

[70] P. Gill, W. Murray, M. Saunders, and M. Wright. Sparse matrix methods in optimization. *SIAM Journal on Scientific and Statistical Computing*, 5:562–589, 1984.

[71] P. Gilmore and C. T. Kelley. An implicit filtering algorithm for optimization of functions with many local minima. *SIAM Journal on Optimization*, 5:269–285, 1995.

[72] D. Goldfarb and J. Reid. A practicable steepest-edge simplex algorithm. *Mathematical Programming*, 12:361–371, 1977.

[73] G. Golub and C. Van Loan. *Matrix computations*. JHU Press, 2012.

[74] J. Gondzio. Stable algorithm for updating dense LU factorization after row or column exchange and row and column addition or deletion. *Optimization*, 23:7–26, 1992.

[75] N. Gould, D. Orban, and Ph. L. Toint. `HSL`: A collection of Fortran codes for large scale scientific computation, Current as of 3 July, 2020. `http://www.hsl.rl.ac.uk/catalogue/la04.html`.

[76] G. A. Gray and T. G. Kolda. Algorithm 856: `APPSPACK` 4.0: Parallel pattern search for derivative-free optimization. *ACM Transactions on Mathematical Software*, 32:485–507, 2006.

[77] J. D. Griffin and T. G. Kolda. Asynchronous parallel hybrid optimization combining DIRECT and GSS. *Optimization Methods & Software*, 25:797–817, 2010.

[78] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tile QR factorization with parallel panel processing for multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[79] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *The International Journal of High Performance Computing Applications*, 29:193–208, 2015.

[80] J. Hall and K. McKinnon. ASYNPLEX, an asynchronous parallelrevised simplex algorithm. *Annals of Operations Research*, 81:27–50, 1998.

[81] N. Hansen. *The CMA Evolution Strategy: A tutorial.* `http://www.lri.fr/~hansen/cmaesintro.html`.

[82] M. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In*, SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2002:1–10, 2002.

[83] T. Hastie, R. Tibshirani, and R. J. Tibshirani. Extended comparisons of best subset selection, forward stepwise regression, and the lasso. *Journal of Statistical Science*, 2020 Accepted.

[84] Lili He, Hongtao Bai, Yu Jiang, Dantong Ouyang, and Shanshan Jiang. Revised simplex algorithm for linear programming on gpus with cuda. *Multimedia Tools and Applications*, 77(22):30035–30050, 2018.

[85] T. Hemker and C. Werner. DIRECT using local search on surrogates. *Pacific Journal of Optimization*, 7:443–466, 2011.

[86] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[87] K. Holmström, A. O. Göran, and M. M. Edvall. *User's Guide for* `TOMLAB` *7*. Tomlab Optimization, Current as of 19 July, 2019. `http://tomopt.com`.

[88] R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. *Journal of the Association for Computing Machinery*, 8:212–219, 1961.

[89] Q. Huangfu and JA Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10:119–142, 2018.

[90] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. *LNCS*, 6140:186–202, 2010.

[91] W. Huyer and A. Neumaier. Global optimization by multilevel coordinate search. *Journal of Global Optimization*, 14:331–355, 1999.

[92] W. Huyer and A. Neumaier. SNOBFIT–Stable noisy optimization by branch and fit. *ACM Transactions on Mathematical Software*, 35:1–25, 2008.

[93] ICL. MAGMA, Current as of 2 June, 2020. `http://icl.cs.utk.edu/projectsfiles/magma/doxygen/`.

[94] L. Ingber. Adaptive simulated annealing (ASA): Lessons learned. *Control and Cybernetics*, 25:33–54, 1996.

[95] W. Jia, K. Shaw, and M. Martonosi. Stargazer: Automated regression-based GPU design space exploration. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 2–13, 2012.

[96] D. R. Jones. The DIRECT global optimization algorithm. In C. A. Floudas and P. M. Pardalos (eds.), *Encyclopedia of Optimization,* Kluwer Academic Publishers, Boston, MA, 1:431–440, 2001.

[97] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Application*, 79:157–181, 1993.

[98] O. Karaca and M. Kamgarpour. Exploiting Weak Supermodularity for Coalition-Proof Mechanisms. In *Proceedings 2018 IEEE Conference on Decision and Control (CDC),* IEEE, Miami Beach, FL, pages 1118–1123, 2018.

[99] C. T. Kelley. *Users Guide for* `IMFIL` *version 1.0*, Current as of 19 July, 2019. `http://www4.ncsu.edu/~ctk/imfil.html`.

[100] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, Piscataway, NJ, USA, 1995.

[101] D. Klabjan, E. Johnson, and G. Nemhauser. A parallel primal–dual simplex algorithm. *Operations Research Letters*, 27:47–55, 2000.

[102] R. Koenker and G. Bassett. Regression quantiles. *Econometrica: journal of the Econometric Society*, 46:33–50, 1978.

[103] R. Koenker and K. Hallock. Quantile regression. *Journal of economic perspectives*, 15:143–156, 2001.

[104] R. Kohavi and G. John. Wrappers for feature subset selection. *Artificial intelligence*, 97:273–324, 1997.

[105] T. G. Kolda, R. M. Lewis, and V. J. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45:385–482, 2003.

[106] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright. Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9:112–147, 1998.

[107] M. Lalami, D. El-Baz, and V. Boyer. Multi GPU implementation of the simplex algorithm. In*, IEEE International Conference on High Performance Computing and Communications,* IEEE Computer Society, Washington, DC, pages 179–186, 2011.

[108] S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In*, Proceedings of the 2001 ACM/IEEE conference on Supercomputing,* IEEE, New York, NY, pages 55–60, 2001.

[109] C. Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1:36–47, 1954.

[110] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *International Conference on Computational Science*, pages 884–892, 2009.

[111] L. Liberti, P. Poirion, and K. Vu. Fast approximate solution of large dense linear programs, 2018.

[112] E. Liberty and M. Sviridenko. Greedy minimization of weakly supermodular set functions. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2017)*, 2017.

[113] I. Loshchilov and F. Hutter. CMA-ES for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269*, pages 1–15, 2016.

[114] Y. Luo and R. Duraiswami. Efficient parallel nonnegative least squares on multicore architectures. *SIAM Journal on Scientific Computing*, 33:2848–2863, 2011.

[115] S. Madougou, A. Varbanescu, C. de Laat, and R. van Nieuwpoort. The landscape of GPGPU performance modeling tools. *Parallel Computing*, 56:18–33, 2016.

[116] B. Mamalis and M. Perlitis. Simplex Parallelization in a Fully Hybrid Hardware Platform. *International Journal of Advanced Computer Science and Applications*, 8:356–365, 2017.

[117] N. Mantel. Why stepdown procedures in variable selection. *Technometrics*, 12:621–625, 1970.

[118] H. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269, 1957.

[119] N. Meinshausen. Relaxed lasso. *Computational Statistics & Data Analysis*, 52:374–393, 2007.

[120] D. Merrill and M. Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 678–689. IEEE, 2016.

[121] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21:1087–1092, 1953.

[122] K. Murty. *Linear programming*. Springer, 1983.

[123] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *The International Journal of High Performance Computing Applications*, 24:511–515, 2010.

[124] M. Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, 1, 2011.

[125] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[126] G. Nemhauser, L. Wolsey, and M. Fisher. An analysis of approximations for maximizing submodular set functions-I. *Mathematical Programming*, 14:265–294, 1978.

[127] A. Neumaier. `MCS`: Global Optimization by Multilevel Coordinate Search, Current as of 19 July, 2019.
`http://www.mat.univie.ac.at/~neum/software/mcs/`.

[128] NVIDIA Corporation. cuSolver, Current as of 24 June, 2020. `https://docs.nvidia.com/cuda/cusolver/index.html`.

[129] NVIDIA Corporation. cuSPARSE, Current as of 24 June, 2020. `https://docs.nvidia.com/cuda/cusparse/index.html`.

[130] NVIDIA Corporation. CUDA, Current as of 25 June, 2020. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[131] N. Nystrom, M. Levine, R. Roskies, and J. Scott. Bridges: A uniquely flexible HPC resource for new communities and data analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure,* Association for Computing Machinery, New York, NY, pages 1–8, 2015.

[132] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, pages 80–113, 2007.

[133] M. Pacula, J. Ansel, S. Amarasinghe, and U. O'Reilly. Hyperparameter tuning in bandit-based adaptive operator selection. In C. Chio and A. Agapitos and S. Cagnoni and C. Cotta and F. Vega (eds.)*, Proceedings of the 2012t European Conference on Applications of Evolutionary Computation,* Springer-Verlag, Berlin, Heidelberg, pages 73–82, 2012.

[134] K. Paparrizos. An infeasible (exterior point) simplex algorithm for assignment problems. *Mathematical Programming*, 51(1-3):45–54, 1991.

[135] J. D. Pintér. *Global Optimization in Action: Continuous and Lipschitz Optimization. Algorithms, Implementations and Applications.* Nonconvex Optimization and Its Applications. Kluwer Academic Publishers, 1995.

[136] J. D. Pintér, K. Holmström, A. O. Göran, and M. M. Edvall. *User's Guide for* `TOMLAB/LGO`. Tomlab Optimization, 2006. `http://tomopt.com`.

[137] T. D. Plantenga. HOPSPACK 2.0 User Manual. Technical Report SAND2009-6265, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2009. `https://software.sandia.gov/trac/hopspack/`.

[138] N. Ploskas and N. Samaras. GPU accelerated pivoting rules for the simplex algorithm. *Journal of Systems and Software*, 96:1–9, 2014.

[139] N. Ploskas and N. Samaras. Efficient GPU-based implementations of simplex type algorithms. *Applied Mathematics and Computation*, 250:552–570, 2015.

[140] M. J. D. Powell. Recent research at Cambridge on radial basis functions. Technical report, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, 1998.

[141] M. J. D. Powell. UOBYQA: Unconstrained Optimization BY Quadratic Approximation. *Mathematical Programming*, 92:555–582, 2002.

[142] M. J. D. Powell. Developments of NEWUOA for minimization without derivatives. *IMA Journal of Numerical Analysis*, 28:649–664, 2008.

[143] M. J. D. Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical report, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, 2009.

[144] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21:703–712, 2002.

[145] B. Ratner. Variable selection methods in regression: Ignorable problem, outing notable solution. *Journal of Targeting, Measurement and Analysis for Marketing*, 18:65–75, 2010.

[146] J. Reid. A sparsity-exploiting variant of the Bartels—Golub decomposition for linear programming bases. *Mathematical Programming*, 24:55–69, 1982.

[147] L. M. Rios and N. V. Sahinidis. Derivative-free optimization: A review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56:1247–1293, 2013.

[148] S. Sakaue. Weak supermodularity assists submodularity-based approaches to non-convex constrained optimization. *Arxiv preprint arXiv*, pages 1–26, 2019.

[149] O. Sarwar, B. Sauk, and N. V. Sahinidis. A Discussion on Practical Considerations with Sparse Regression Methodologies. *Statistical Science*, Submitted, 2020.

[150] B. Sauk, N. Ploskas, and N. V. Sahinidis. A GPU parallel block-LU update. In preparation, 2020.

[151] B. Sauk, N. Ploskas, and N. V. Sahinidis. GPU paramter tuning for tall and skinny dense linear least squares problems. *Optimization Methods and Software*, 35:638–660, 2020.

[152] B. Sauk and N. V. Sahinidis. Backward stepwise elimination: Approximation guarantee, a batched GPU algorithm, and empirical investigation. *Journal of the American Statistical Association*, Submitted, 2020.

[153] B. Sauk and N. V. Sahinidis. HybridTuner: Tuning with hybrid derivative-free optimization initialization strategies. *ACM Transactions on Architecture and Code Optimization (TACO)*, Submitted, 2020.

[154] M. Saunders. The complexity of LU updating in the simplex method. *The complexity of computational problem solving', University Press, Queensland*, pages 214–230, 1973.

[155] R. Schreiber and C. Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific Computing*, 10:53–57, 1989.

[156] W. Shu. Parallel implementation of a sparse simplex algorithm on MIMD distributed memory computers. *Journal of Parallel and Distributed computing*, 31:25–40, 1995.

[157] B. O. Shubert. A sequential method seeking the global maximum of a function. *SIAM Journal on Numerical Analysis*, 9:379–388, 1972.

[158] R. L. Smith. Efficient Monte Carlo procedures for generating points uniformly distributed over bounded regions. *Operations Research*, 32:1296–1308, 1984.

[159] J. Snoek, H. Larochelle, and R.P. Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira and C.J.C. Burges and L. Bottou and K.Q. Weinberger (eds.)*, Proceedings of the 25th International Conference on Neural Information Processing Systems,* Curran Associates Inc., Red Hook, NY, pages 2951–2959, 2012.

[160] J. C. Spall. Chapter 7: Simultaneous Perturbation Stochastic Approximation. In *Introduction to stochastic search and optimization: Estimation, simulation, and control.* Wiley-Interscience, 2003.

[161] D. Spampinato and A. Elstery. Linear optimization on modern GPUs. In*, IEEE International Symposium on Parallel & Distributed Processing,* IEEE, Red Hook, NY, pages 1–8, 2009.

[162] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis,* Association for Computing Machinery, New York, NY, pages 35–46, 2011.

[163] C. Ţăpuş, I. Chung, and J. Hollingsworth. Active harmony: Towards automated performance tuning. In *Proceedings of the ACM/IEEE Conference on Supercomputing,* IEEE Computer Society Press, Washington, DC, USA, pages 1–11, 2002.

[164] M. Tartara and S. Reghizzi. Continuous learning of compiler heuristics. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9:1–25, 2013.

[165] M. Thomadakis and J. Liu. An efficient steepest-edge simplex algorithm for SIMD computers. In*, Proceedings of the 10th international conference on Supercomputing* ICS, pages 286–293, 1996.

[166] R. Tibshirani. Regression shrinkage and selection via the lasso. *Royal Statistical Society*, 58:267–288, 1996.

[167] R. Tibshirani. Degrees of freedom and model search. *Statistica Sinica*, 25:1265–1296, 2015.

[168] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36:232–240, 2010.

[169] V. J. Torczon. On the convergence of multidirectional search algorithms. *SIAM Journal on Optimization*, 1:123–145, 1991.

[170] V. J. Torczon. On the convergence of pattern search algorithms. *SIAM Journal on Optimization*, 7:1–25, 1997.

[171] J. Towns, T. Cockerill, M. Dahan, I. foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. Peterson, R. Roskies, J.R. Scott, and N. Wilkens-Diehr. XSEDE: Accelerating Scientific Discovery. Computing in Science & Engineering. *Computing in Science & Engineering*, 16:62–74, 2014.

[172] A. I. F. Vaz. `PSwarm` Home Page, Current as of 19 July, 2019. `http://www.norg.uminho.pt/aivaz/pswarm/`.

[173] A. I. F. Vaz and L. N. Vicente. A particle swarm pattern search method for bound constrained global optimization. *Journal of Global Optimization*, 39:197–219, 2007.

[174] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *2008 SC-International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2008.

[175] P. D. Vouzis and N. V. Sahinidis. GPU-BLAST: Using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27:182–188, 2011.

[176] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, 2005.

[177] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.

[178] Z. Xianyi, W. Qian, and Z. Yunquan. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 684–691, 2012.

[179] T. Yuki and L. N. Pouchet. PolyBench/C 4.2.1, Current as of 1 June, 2020. `https://www.cs.colostate.edu/~pouchet/software/polybench/polybench-fortran.html`.

[180] Y. Zhang, P. Vouzis, and N. V. Sahinidis. GPU simulations for risk assessment in CO2 geologic sequestration. *Computers & Chemical Engineering*, 35:1631–1644, 2011.