To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–23

GPU parameter tuning for tall and skinny dense linear least squares problems

Benjamin Sauk[a], Nikolaos Ploskas[a] and Nikolaos Sahinidis[a]*

[a]*Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

Linear least squares problems (LLSPs) routinely arise in many scientific and engineering problems. One of the fastest ways to solve LLSPs involves performing calculations in parallel on graphics processing units (GPUs). However, GPU algorithms are typically designed for one GPU architecture and may be suboptimal or unusable on another GPU. To design optimal algorithms for any GPU with little need for modifying code, tunable parameters can simplify the transition of GPU algorithms to different GPU architectures. In this paper, 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. Computational experiments show that both DFO and SO can be effective tools for determining optimal tuning parameters that can 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 were able to identify optimal parameters using an order of magnitude fewer simulations than with direct enumeration.

Keywords:
Linear least squares; Derivative-free optimization; Graphics processing unit; Parallel computing; Parameter tuning

1. Introduction

Numerical solvers on graphics processing units (GPUs) are typically designed for one particular GPU architecture [45], and may be suboptimal or even unusable on another one [39]. 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) [59] and simulation optimization (SO) [6] 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

*Corresponding author. Email: sahinidis@.cmu.edu

processing, parameter estimation, and machine learning, including best subset selection and the lasso [16, 24]. 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 [29], or more expensive computational costs, respectively [24]. 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 [44]. 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 [51]. 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 [49, 50]. 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 [64]. 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 [5, 43, 68]. 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 [43]. 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 [10] 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 [59] 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, 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 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 4, we describe different types of DFO and SO algorithms that were used in our experiments to determine optimal GPU tuning parameters. In Section 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 6.

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 [4].

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 [7]. 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 [60], it is possible to delay and apply many updates simultaneously with matrix-matrix multiplication.

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 [8]. 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 [25] 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 [8]. 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.

2.3 *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 [7]. 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 [19], is designed to accelerate dense linear algebra routines by assigning sequential tasks to the CPU and parallelizable tasks to the GPU [64]. 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 [37]. 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 [8]. 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 [50]. One advantage of cuSolverDN is that NVIDIA has knowledge of how their software and hardware interact, allowing them to optimize their solver.

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 [8]. 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 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 [13].

Table 1.: QR libraries used in comparative experiments

Solver	Computing environment	BLAS library
cuSolverDN v7.5 [50]	GPU only	cuBLAS v7.5 [49]
LAPACK v3.6.1 [7]	CPU only	OpenBLAS v0.2.18 [69]
MAGMA v2.1 [64]	Hybrid	cuBLAS v7.5
PLASMA v2.8.0 [30]	Multicore CPU	Intel MKL v16.0.3

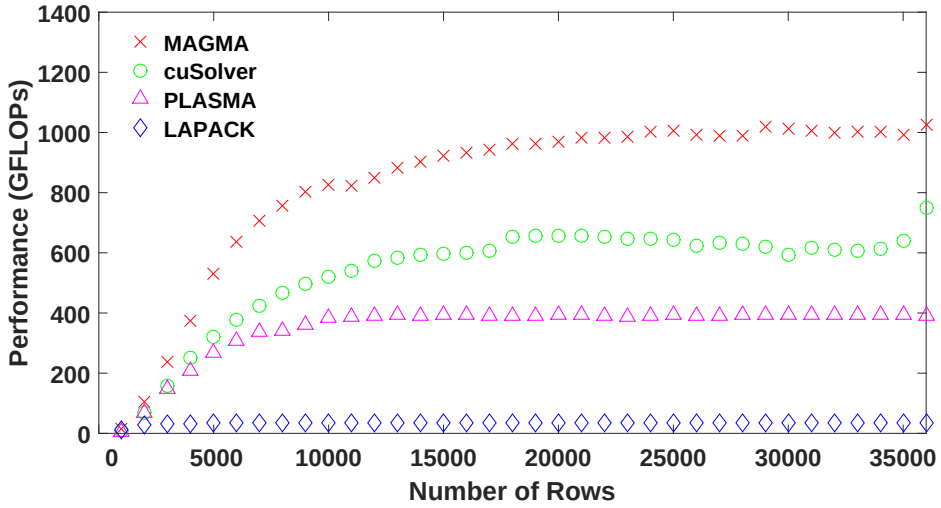

Figure 1.: Performance of four different LLSP solvers for square matrix problems

3.1 *Square matrices*

We factor square matrices ranging from 1000 to 36000 rows by increments of 1000. Figure 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 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

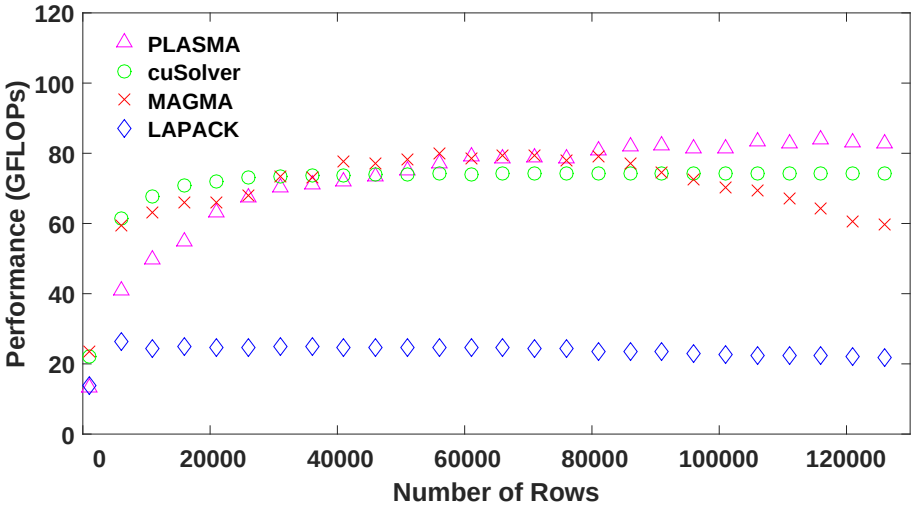

Figure 2.: Performance of four different LLSP solvers for TS matrix problems

GPU solvers because of how efficiently the GPU solvers are able to factor square dense problems.

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 [21, 27]. 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, 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 [8]. 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, 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.

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.

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 [6, 59]. 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 [59], and to [6] for a more thorough analysis of SO solvers. Table 2 provides a listing of the DFO and SO solvers investigated in this paper.

Table 2.: DFO and SO solvers used in this paper

Solver	Type
ASA† [38]	DFO, global, stochastic
BOBYQA† [58]	DFO, local, model-based
CMA-ES† [31]	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† [20]	DFO, local, model-based
FMINSEARCH† [42]	DFO, local, direct
GLOBAL† [22]	DFO, global, stochastic
HOPSPACK† [54]	DFO, local, direct
IMFIL† [41]	DFO, local, model-based
MCS† [48]	DFO, global, deterministic
NEWUOA† [57]	DFO, local, model-based
NOMAD† [2]	DFO, local, direct
PRAXIS† [15]	DFO, local, direct
PSWARM† [67]	DFO/SO, global, stochastic
SID-PSM† [23]	DFO, local, direct
SNOBFIT† [36]	DFO/SO, global, deterministic
TOMLAB/CGO† [33]	DFO, global, deterministic
TOMLAB/GLB† [40]	DFO, global, deterministic
TOMLAB/GLC† [40]	DFO, global, deterministic
TOMLAB/GLCCLUSTER† [40]	DFO, global, deterministic
TOMLAB/LGO† [53]	DFO, global, stochastic
TOMLAB/MSNLP† [33]	DFO, global, hybrid
TOMLAB/RBF† [55]	DFO, global, deterministic
TOMLAB/GLCDIRECT* [33]	DFO, global, deterministic
TOMLAB/MIDACO* [33]	DFO, global, stochastic
TOMLAB/GLCFAST* [33]	DFO, global, deterministic
TOMLAB/GLCSOLVE* [33]	DFO, global, deterministic
TOMLAB/GLCCLUSTER* [33]	DFO, global, deterministic
SPSA BASIC† [63]	SO, local, stochastic approximation
SPSA Second Order† [63]	SO, local, stochastic approximation
STRONG† [18]	SO, local, response surface, trust region
SNM* [17]	SO, global, direct search

† solver accepts continuous variables only

* solver accepts continuous and integer variables

4.1 *DFO local search methods*

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 [34]. 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 [9, 47, 66]. Originally, direct methods were used to solve difficult problems without any formal termination or convergence proofs [59]. As the field has developed, under certain assumptions, a few different direct methods can be shown to converge to a stationary point [65].

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 [59]. 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 [28, 56].

4.2 *DFO global search methods*

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 [61]. 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 [40]. 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 [35]. 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 [52]. 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 [11], pattern search [14], and optimization by branch-and-fit [36].

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 [12]. Some popular methods include hit-and-run algorithms [62], simulated annealing [46], genetic algorithms [32], and particle swarm optimization [26]. Many global derivative-free optimization solvers combine deterministic and stochastic methods into hybrid algorithms.

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 [6]. 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 [63].

Direct search methods optimize by performing a sequential examination of points generated by some strategy [34]. 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 [17].

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, sim-

ulated 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 probability 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 [6].

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.

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 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

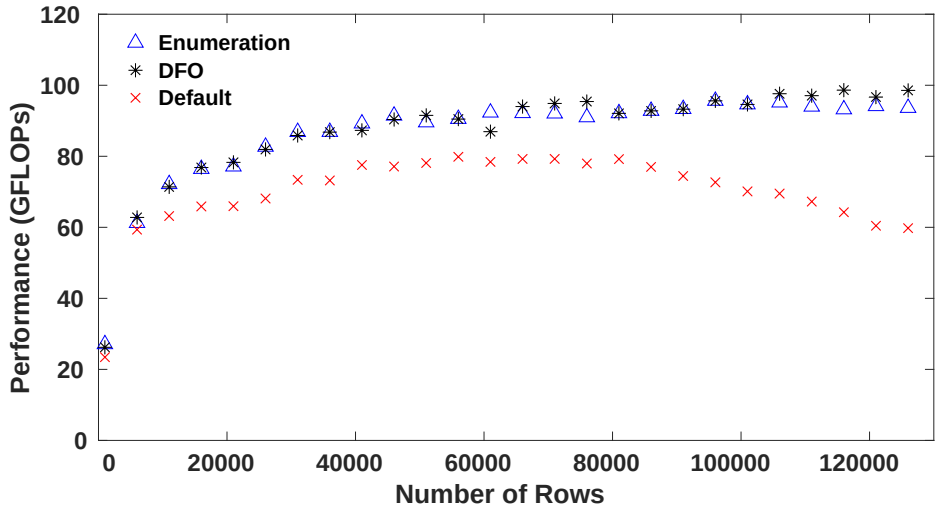

Figure 3.: MAGMA's QR factorization performance with different tuning parameters

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 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 3, we see that for smaller problems there is not much benefit in optimizing 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 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.

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

Table 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

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 4, the DFO solvers used were compared to demonstrate how well each performed for all of the TS problems. The columns in Table 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 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

Table 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

maximizing over are not well-behaved concave functions, as shown in Figure 4. Figure 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.

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 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.

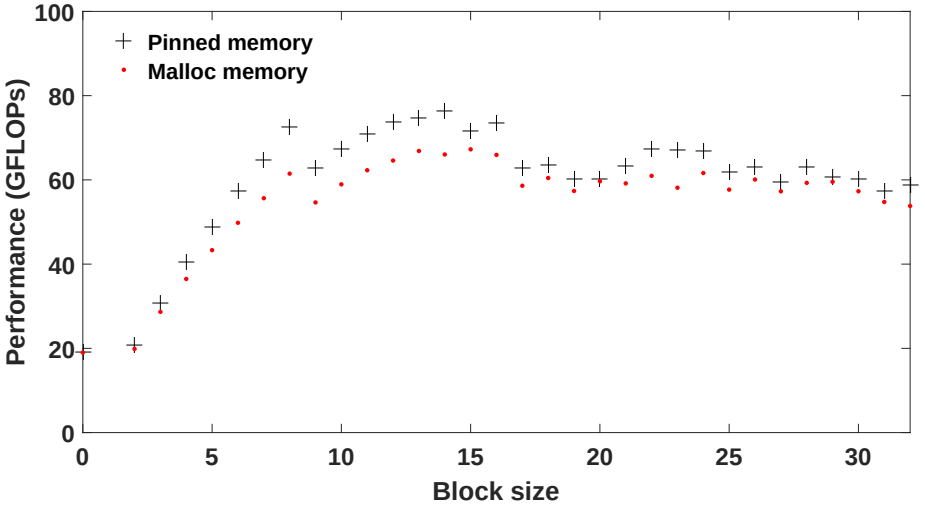

Figure 4.: QR factorization performance for one problem with different options selected

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 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

Table 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

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 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.

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 per-

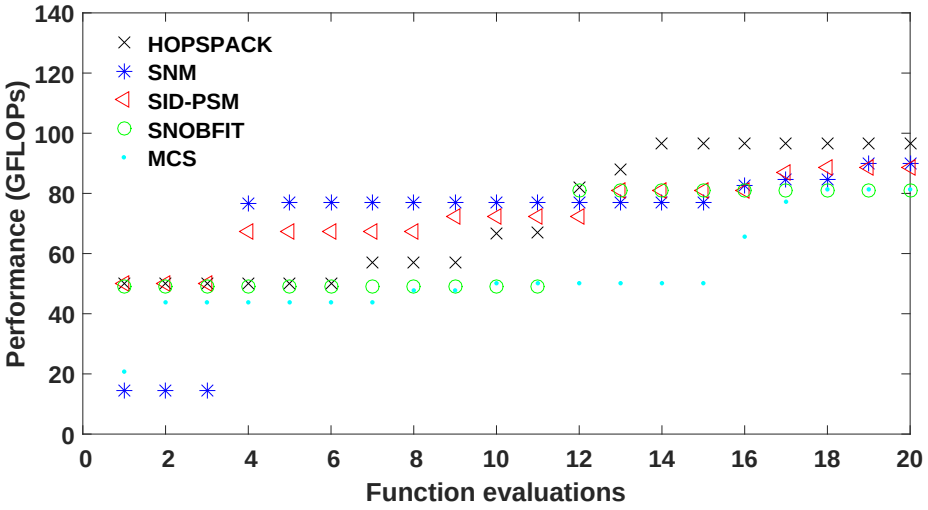

Figure 5.: Tracking the best performance found as the number of function evaluations given to the different solvers increases for QR factorization of a 121000 by 500 matrix

formance than PLASMA and cuSolver, when tuned with DFO, MAGMA was able to outperform all of the other solvers. As seen in Figure 6, this parameter tuning made MAGMA the best performing LLSP solver over the entire range of tall and skinny matrices that we considered.

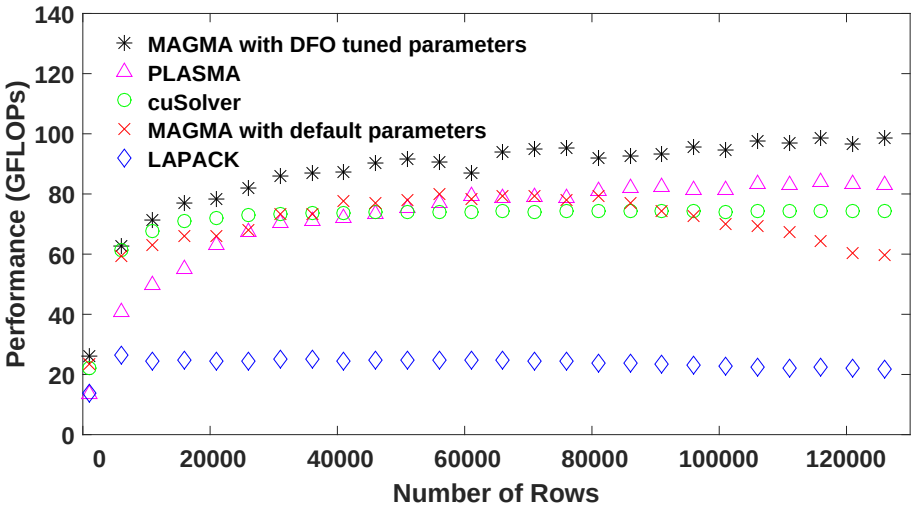

Figure 6.: Performance of four QR solvers, compared against the performance of MAGMA with DFO-determined parameters

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.

Acknowledgments

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.

We also gratefully acknowledge the support of the NVIDIA Corporation with the donation of the NVIDIA Tesla K40 GPU used for this research.

Disclaimer

References

[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. `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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] C. Audet and D. Orban. Finding optimal algorithmic parameters using derivative-free optimization. *Society for Industrial and Applied Mathematics*, 17:642–664, 2001.

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

[23] 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.

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

[25] 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.

[26] 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.

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

[28] 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.

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

[30] 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.

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

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

[33] K. Holmström, A. O. Göran, and M. M. Edvall. *User's Guide for* `TOMLAB` *7*. Tomlab Optimization, 2010. `http://tomopt.com`.

[34] 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.

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

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

[37] ICL. MAGMA, Current as of 6 July, 2017. `http://icl.cs.utk.edu/projectsfiles/magma/doxygen/index.html`.

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

[39] 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.

[40] 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.

[41] C. T. Kelley. *Users Guide for* `IMFIL` *version 1.0*. `http://www4.ncsu.edu/~ctk/imfil.html`.

[42] 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.

[43] 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.

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

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

[46] 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.

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

[48] A. Neumaier. `MCS`: Global Optimization by Multilevel Coordinate Search. `http://www.mat.univie.ac.at/~neum/software/mcs/`.

[49] NVIDIA Corporation. cuBLAS, Current as of 28 December, 2016. `http://docs.nvidia.com/cuda/cublas/#axzz4SZ3ssQJO`.

[50] NVIDIA Corporation. cuSolver, Current as of 28 December, 2016. `http://docs.nvidia.com/cuda/cusolver/#axzz4SZ3ssQJO`.

[51] 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.

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

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

[54] 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/`.

[55] 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.

[56] M. J. D. Powell. UOBYQA: unconstrained optimization by quadratic approximation. *Mathematical Programming*, 92:555–582, 2002.

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

[58] 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.

[59] 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.

[60] 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.

[61] B. O. Shubert. A sequential method seeking the global maximum of a function. *SIAM Journal on*

Numerical Analysis, 9:379–388, 1972.

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

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

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

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

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

[67] A. I. F. Vaz. `PSwarm` Home Page. `http://www.norg.uminho.pt/aivaz/pswarm/`.

[68] 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.

[69] 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.