GPU-based and Multi-Core Linear Programming Optimization

Matthew Hall and Chris Misa

University of Oregon, CIS 631

Abstract. In this paper, we present a parallel implementation of the revised simplex method using two parallel processing methods; a multi-core method using OpenMP, and a GPU method using CUDA. To this end, we develop a robust implementation of revised simplex with support for pluggable backends using operations from different architectures, be it single-core, multi-core or GPU. We compare the speedup of our parallel implementations against a serial implementation of the same operations, and a cuBLAS-accelerated implementation, and present microbenchmarks for the various operation primitives required by revised simplex. We test our code using Linear Programming problem definitions from the Netlib repository and show that our implementations improve the runtime performance of medium and large scale problems, with increasing performance benefit for larger problems. We also see that for the largest problems, the cuBLAS implementation is 19 × faster than our CUDA kernels, solving a problem in 13 s that took 212 s for our CUDA kernel, and more than one hour for our single-threaded implementation.

1 Introduction

Linear Programming (LP) is a widely used tool for solving optimization problems. Some common use cases include scheduling, supply chain management, production planning, network flow, and operations research in general. It is commonly applied in different research domains, including networked systems. Originally developed in the 1800s, it has blossomed with the development of computing, as faster computers, larger memory capacity, and multi-processing have enabled LP to be applied to larger and larger problems. For instance, Google and Microsoft, both rely on linear programming as the core component of their traffic engineering systems, which continuously serve content to billions of people daily [17, 18].

The simplex method for linear programming is one of the most commonly deployed optimization algorithms. Although it has worst-case exponential time complexity (for the number of constraints), simplex often finds an optimal solution on polynomial time [29]. Although polynomial-time algorithms are more attractive than exponential, they can still prove to be intractable for large problems.

Parallel to the growth of problem size and complexity, computer architectures have advanced as well. Multi-core processors are now ubiquitous in commodity and high-performance computers. Graphics processors, with thousands of cores, are pervasive as well. The paradigm shift towards many-core architectures compels developers to rewrite old single-core algorithms for these new multi-core systems if they expect their programs to speed up year after year and to solve larger and more complex problems.

Given that linear programs, although powerful, have their limits of computational tractability, and that more focus must be given to the parallel programming paradigm to make the best use of the computational resources available today, we aim to develop parallel programming implementations of the revised simplex method, thereby solving optimization problems more efficiently on modern computing systems.

In particular, we explore two avenues of parallelization: multi-core, and GPU. We relate our experience with writing these algorithms in this paper and highlight the performance boosts that we observed in the single-core (serial) version of the algorithm as well as the multi-core (OpenMP) and GPU (CUDA) implementations. We also compare our CUDA implementation with a high-performance library, *cuBLAS*. The key result we present is that *problem size has a critical effect on the ability of parallelization to achieve better performance*, regardless of the chosen avenue of parallelization. In general, we note that problems with hundreds of variables and constraints are able to benefit from parallel implementations, while smaller problems are more efficient to solve using serial code.

The remainder of this report is organized as follows: In §2 we discuss the simplex and revised simplex algorithms; in § 3 we discuss our implementation of those algorithms and our generic backend interface. § 4 details the performance results of our

OpenMP and CUDA backends on microbenchmarks and selected Netlib problems. § 5 touches on related work. Finally, we give concluding remarks in § 6.

Aside: In our original project proposal, we planned to develop parallel implementations of an Integer Programming optimization solver. Through deeper study, we realized that LP optimization solvers were required at the core of most common algorithms for the IP problem. Due to time constraints, we decided to focus the current effort instead on developing an LP implementation. Our current implementation is highly flexible and performant on large problems and lays the groundwork for future IP and Mixed-IP optimization.

2 Background

An LP optimization problem consists of three main parts: (1) the objective functions, expressed as a linear combination of a set of variables; (2) the optimization sense of the objective function, expressed as minimize or maximize; and (3) a set of constraints on the variables. Often, LP problems arise in practice with tens to hundreds of variables and complex, heterogeneous sets of constraints. For example, Gardner et al. [14] cast an audit staff planning problem as an LP with 153 constraints, including 95 equalities, 16 less-thans, and 42 greater-thans, on 308 variables. To simplify the task of specifying LPs, the common MPS format [2, 3] allows the specification of equalities and inequalities over the variables, as well as bounds on each individual variable and ranges in which to apply the given equalities and/or inequalities. LP solvers can easily convert such heterogeneous inputs into the less relaxed forms required by LP optimization algorithms.

To simplify the mathematics, most LP optimization algorithms assume the input is given in a concise mathematical description called standard form. Following loosely the notation of Ralphs [25], the standard form for expressing an LP problem is as follows:

$$\min_{x \in \mathcal{F}} c^{\mathrm{T}} x \tag{1}$$

$$\text{s.t. } \mathcal{F} = \{ x \in \mathbb{R}^n \mid Ax \leq b \ , \ x \geq 0 \ \} \tag{2}$$

where $A \in \mathbb{R}^{m \mathrm{x} n}$ and $b \in \mathbb{R}^m$ describe the constraints on the variables $x_1, \ldots, x_n$ and $c \in \mathbb{R}^n$ specifies the objective function. Clearly, the heterogeneous types of constraints found in the wild can easily be transformed into standard form by replacing equalities with two inequality constraints, negating greater-than constraints, and possibly negating the objective function. In the last cast, the computed optimal objective value must also be negated before reporting. A final caveat is that unbounded x values must be replaced with a difference between two, new positive variables. While these transformations are mathematically simple, they can result in a significant expansion of constraints and variables. For example, the audit staff planning LP mentioned above actually has twice as many constraints in standard form. Finally, transformations are required to remove redundant or pairwise linearly dependent constraints or variables as will become apparent in section 2.1.

In addition to standard form, several other key terms have developed to facilitate the discussion of LP optimization algorithms. A setting of x which satisfies $x \in \mathcal{F}$ is known as a *feasible* solution and a setting of x which minimizes $c^{\mathrm{T}} x$ is known as an *optimal* solution. The goal of an LP optimization algorithm is then to find the optimal solution given the objective function and constraints. Additionally, LP optimization algorithms must be able to identify two conditions where such an optimal solution does not exist. First, it could be the case that the objective functions can be decreased without bound, in which case the LP is said to be *unbounded*. Finally, the constraints could be given such that no feasible solution exists (e.g., $x \geq 3$ and $x \leq 2$) in which case, the LP is said to be *infeasible*.

Most performant LP optimization solutions are built on Dantzig's original simplex method [11]. At a high level, this method identifies that any optimal solution will fall on the edge of the n-dimensional polyhedron defined by $\mathcal{F}$. The method iteratively explores the convex hull of $\mathcal{F}$, going from point to point, until the optimal solution is found. The complexity of this algorithm is typically measured in the number of iterations taken for a given problem as a function of problem dimensionality. While the worst-case time complexity of the simplex method is known to be exponential, for a wide range of practical optimization problems, it executes in near-linear time [26]. Probabilistic analysis [7, 27] demonstrates the average case time complexity is, in fact, linear. Several other theoretic results support the simplex method's utility when the dimension is fixed [12, 23].

Another key feature of Dantzig's method is its ready interpretation in terms of common linear algebra operations, often known as the *revised simplex method*. The textbook formulation of the original simplex method is given in terms of updates to constraints and the objective function. In each iteration, these equations are rearranged such that particular *basic* variables are isolated in terms of the

other *nonbasic* variables. This naive form requires updating each coefficient (each entry in A) in each iteration which leads to slow memory-bound implementations in practice. The key insight behind the revised simplex method is that the coefficients of the basic variables at any iteration form a basis for the right-hand-side of the constraints (i.e., b). The iterative selection of basic variables, then, can be made by simply replacing columns of this basis and computing its inverse. These operations are much more efficient on modern computer architectures due to their higher arithmetic intensity. We use the word simplex to refer to the revised simplex method, and not Dantzig's original algorithm in the remainder of this paper.

2.1 The Revised Simplex Method

To execute the revised simplex method, the LP must first be converted from standard form to slack form. This essentially converts all the inequalities to equalities by adding non-negative "slack" variables s_i such that

$$\sum_{j=1}^{n} a_{ij}x_j + s_i = b_i \ , \ \forall i \in \{1, \ldots, m\}.$$

For simplicity of notation, the following discussion assumes the LP has already been converted into slack form by replacing A with AI, where I is the m-element identity matrix, and n with $m + n$. The feasible region $\mathcal{F}$ then also becomes $\{x \in \mathbb{R}^n \mid Ax = b \ , \ x \geq 0 \}$. Since $n > m$ and we assume the columns of A are pairwise linear independent, this system of equalities clearly has infinite solutions.

As in the original simplex method, the revised simplex method considers a sequence of solutions to this system of equalities, moving from less optimal to more optimal solutions based on a set of heuristics. To limit the infinite solution space, the revised simplex method only considers *basic* solutions. A basic solution is formed by extracting a basis from the columns of A and setting the variables that correspond to the other, nonbasic columns to zero. The system can then be solved for the values of the basic variables by inversion and the value of the objective function can be determined. These solutions correspond to points of the n-dimensional polyhedron considered in the geometric interpretation of the simplex method.

Due to the nonnegativity constraint on the x_is and the fact than the b_is may still be negative, *basic* solutions do not always correspond to *feasible* solutions. Therefore, implementations of the simplex method must proceed in two phases: first to find a *basic feasible* solution, then to refine the basic feasible solution into an *optimal* solution. The mechanism traditionally used to accomplish this is to introduce m new nonnegative *artificial* variables with negative coefficients in the constraint matrix A (e.g., augment A with $-I$). The *phase one* objective function then minimizes the sum of these artificial variables. If the sum can be driven to 0, then a basic feasible solution has been found. Otherwise, the LP is determined to be infeasible. While the mathematical correctness of this technique follows easily, the introduction of new variables again increases the problem complexity.

Phase One Algorithm: The current work leverages more recent theoretical developments which provide an *artificial-free* method for finding the *basic feasible* solution [20]. As the name suggests, this method does not require any additional variables beyond the slack form. The key insight is that only the rows of A where the corresponding entry in b are negative need to be considered in the phase one objective function. The same semantics around artificial variables apply to these constraints: if they can all be driven to zero, a *basic feasible* solution has been found, otherwise, the LP is infeasible.

Algorithm 1 shows pseudocode for the phase-one simplex method. We adopt the convention that $A_{\mathcal{S}}$ represents the columns of matrix A specified by the index set S and $c_{\mathcal{S}}$ represents the entries of vector c specified by the index set S. In this notation, β represents the basic solution gathered by setting all nonbasic variables to 0 and the columns of α hold the coefficients of the nonbasic variables with respect to the current choice of basis B, sometimes referred to as the tableau of A. The key steps are in lines 12 and 13. In line 12, the nonbasic variable with maximal impact on the phase one objective function is chosen to be pivoted into the basis only taking into account the constraints that violate solution feasibility (where $b_i < 0$). Next, in line 13, a column is chosen to be removed from the basis using a variation of the classic minimum ratio test. In particular, columns are only removed when the basic solution is zero if the corresponding tableau entry is positive.

Phase Two Algorithm: The second phase of the revised simplex method, as shown in algorithm 2, starts from a basic feasible solution and iteratively refines the value of the original objective function by pivoting basic and nonbasic variables. Each iteration chooses a nonbasic variable to enter the basis with maximal cost improvement as computed in line 2. If none of the nonbasic variables can improve the cost, the current basic solution is optimal and the algorithm terminates. The proof of this is well known

Algorithm 1 Revised Simplex Method Phase 1

Require: A, b, m, n
1: $A \leftarrow [A I]$ ▷ Augment A with slack variables
2: $\mathcal{B} \leftarrow \{n+1, \ldots, n+m+1\}$, $B \leftarrow A_{\mathcal{B}}$ ▷ Choose initial basis
3: $\mathcal{R} \leftarrow \{1, \ldots, n\}$ ▷ Remaining variables are nonbasic
4: $\beta \leftarrow B^{-1} b$, $\alpha \leftarrow B^{-1} A_{\mathcal{R}}$
5: **if** all $\beta_i \geq 0$ **then**
6: **continue** to phase two ▷ Found *basic feasible* solution
7: **end if**
8: $w_j \leftarrow \sum_i \alpha_{i,j}$ s.t. $\beta_i < 0 \quad \forall j \in \mathcal{R}$
9: **if** all $w_j \geq 0$ **then**
10: **return** ▷ LP is *unfeasible*
11: **end if**
12: $q \leftarrow \text{argmin}\{w_j\}$
13: $p \leftarrow \text{argmin}\{\beta_i / \alpha_{i,q} \mid (\beta_i < 0 \text{ and } \alpha_{i,q} < 0) \text{ or } (\beta_i \geq 0 \text{ and } \alpha_{i,q} > 0)\}$
14: Pivot q into $\mathcal{B}$ and p into $\mathcal{R}$, update $B \leftarrow A_{\mathcal{B}}$
15: **goto** line 4

Algorithm 2 Revised Simplex Method Phase 2

Require: $A, b, c, m, n, \mathcal{B}$ ▷ Assuming $A_{\mathcal{B}} x = b$ has a solution where all $x_i \geq 0$
1: $\beta \leftarrow B^{-1} b$, $z \leftarrow c_{\mathcal{B}}^{\mathrm{T}} \beta$, $\pi^{\mathrm{T}} \leftarrow c_{\mathcal{B}}^{\mathrm{T}} B^{-1}$
2: $w \leftarrow c_{\mathcal{R}} - \pi^{\mathrm{T}} A_{\mathcal{R}}$ ▷ Compute the cost improvement for nonbasic variables
3: **if** all $w_j \geq 0$ **then**
4: **return** ▷ No w_j can improve the cost so the current basic solution is optimal
5: **end if**
6: $q \leftarrow \text{argmin}\{w_j\}$
7: $\alpha_{\cdot,q} \leftarrow B^{-1} A_{\cdot,q}$ ▷ Calculate the qth column of the tableau
8: **if** all $\alpha_{i,q} \leq 0$ **then**
9: **return** ▷ The cost can be reduced indefinitely by increasing x_q
10: **end if**
11: $\theta \leftarrow \min\{\beta_i / \alpha_{i,q} \mid \alpha_{i,q} > 0\}$, choose p s.t. $\theta = \beta_p / \alpha_{p,q}$
12: Pivot q into $\mathcal{B}$ and p into $\mathcal{R}$, update B by replacing the p-th column with $A_{\cdot,q}$, and update $\beta_p \leftarrow \theta, \beta_i \leftarrow \beta_i - \theta \alpha_{i,q}$ for $i \neq p$
13: $z \leftarrow z - \theta w_q$ ▷ Update the current objective function value
14: **goto** line 1

and outside the scope of this work (see, e.g., chapter 29 of [9]).

Next, in line 7, this phase calculates the coefficients of the selected column with respect to the current choice of basis. These coefficients show how much increasing the qth variable will decrease the variables in the current basis. If increasing the qth variable actually *increases* the current basic variables, i.e., its coefficients are all nonpositive, the LP is unbounded. Otherwise, in line 11, the algorithm chooses the basic variable which reaches zero first (as the qth nonbasic variable is increased) to pivot out of the basis. As in the phase-one algorithm, the last steps (lines 12 and 13) update the basic and nonbasic variables with a minimum number of operations.

Parallel Implementation: In addition to the simplicity of update operations, the revised simplex algorithm also lends itself somewhat to parallelization. The iterations of the algorithms must be executed in sequence with data synchronization between each iteration. However, the individual operations that must be executed in a single iteration mirror common linear algebra functions which have well-known parallel implementations. In particular several prior efforts explore revised simplex implementation on multi-core [5, 22] and GPU [6, 13, 28] systems. The key challenges through these efforts are in finding efficient techniques for data movement and synchronization between operations. Additionally, the revised simplex method is highly sensitive to the numeric stability of each operation. In particular, the reordering of operations inherent in many parallel reduction implementations (e.g., for the min operations), can have a significant impact on the ability of the algorithm to terminate properly.

3 Implementation

In order to support diverse parallel implementations on a common algorithm, we developed an abstract interface for the required data structures and operations. In §3.1 we describe some of the particular choices made in breaking up the algorithms described in §2 and §3.2 we describe technical details of our implementation including references to source files of interest.

3.1 Motivation

Many of the steps required by the revised simplex algorithm translate directly into common linear algebra operations, however, some steps involve more complex conditional and reduction logic. Our abstract interface reflects this diversity of operations. Rather than attempting to create a general linear algebra solution, we chose to implement only those operations required by the revised simplex algorithms. We additionally identified several key operations that can benefit from parallel implementation but do not fit neatly into traditional linear algebra formulations. As discussed below, these operations

correspond to lines 8 and 13 of algorithm 1 and line 11 of algorithm 2. We also implemented a combined `min` and `argmin` operation which returns the minimum element and its index as required at several points in both phases.

colsum_ltz: This operation, required by line 8 of algorithm 1, computes the sum of the columns of the given matrix, including only the rows where an auxiliary vector (in this case β) is less than zero. We could have implemented this operation by extracting the rows of A corresponding to the nonzero entries in β, transposing the resulting matrix, and using a matrix-vector multiplication with the all-ones vector. However, we anticipated the overheads of two extra function calls as well as the temporary storage and data movement would incur significant overhead. Moreover, this operation lends itself well to implementation as a single parallel pass over the data as each thread can inspect the value of β_i for their particular element independently before embarking on the reduction operation between threads. For example, in the CUDA implementation, this operation is implemented as a single kernel invocation.

phase1_min_ratio: This operation, required by line 13 of algorithm 1, simultaneously computes an element-wise division between two vectors and finds the minimum element of the result. The trick is that only elements where β_i and $\alpha_{i,q}$ are less than zero or elements where β_i is greater than or equal to zero and $\alpha_{i,q}$ is greater than zero should be considered for the minimum. At first glance, it may seem like all elements where the quotient is nonnegative could be considered for the minimum, in which case this could be implemented as three operations: divide, remove negative values, and compute the minimum. However, this would wrongly include elements where β_i is zero but $\alpha_{i,q}$ is *less than zero*. These difficulties along with the fact that these combined operations are easily parallelizable lead us to implement `phase1_min_ratio` as a single operation. In particular, threads can perform the division and contribute to the minimum reduction *only if* the predicate on α and β is met for each particular element. This also reduces the number of expensive division operations performed and may lead to increased performance in hyper-scalar architectures.

phase2_min_ration: This operation, required by line 11 or algorithm 2, is closely related to the phase one minimum ratio test described above. However, only entries where $\alpha_{i,q}$ is strictly greater than zero should be considered. While this could be achieved by extracting the positive elements of $\alpha_{\cdot,q}$, extracting the corresponding elements of β, performing the element-wise division, and taking the minimum, we again reasoned that the overheads, especially in terms of extra copies, would merit the development of a single operation. It might also be conceivable to implement a generic predicated divide and minimum operation which could be shared between both phase's minimum ratio tests, but we felt the overheads of designing and implementing a generic predicate expression language were too great compared with the lesser burden of maintaining two similar operations across different backends.

Another advantage of encapsulating the minimum ratio test for both phases in a single operation is that multiple variants of the artificial-free phase 1 and phase 2 revised simplex methods can be explored with minimal modification. Typically these variants differ only in the heuristics for how to choose the entering and leaving variables which can be (partly) controlled through the minimum ratio test. We hope to explore these different possibilities in future work.

3.2 Implementation Details

As mentioned previously, our implementation strategy follows strict encapsulation of the operations and data structures required for the revised simplex method (see the specification in `include/la.h`). These operations and data structures are implemented and compiled in independently-linked object files allowing for independence between libraries. For example, if the proprietary CUDA libraries are not installed on a particular system, the rest of the code may still be used out of the box. The core revised simplex method is implemented in the `Revised_Simplex()` procedure (see `src/revised_simplex.c`) and makes no assumptions about the underlying data types or implementations, beyond what is defined in the specification. We also developed simple unit tests for sanity (see `src/la_test.c`) and a more performance-oriented benchmarking system (see `src/benchmark.c`) which both rely on this generic interface to apply common tests to each backend developed.

Backends are implemented in single C source files and follow the naming convention `src/la_*`, where the `*` represents the name of the backend (e.g., the OpenMP implementation is in `src/la_omp.c`). Each backend implementation populates a structure of function pointers through the `*_set_ops()` procedure. While the use of function pointers may have some negative performance implications, we contend that the number of operations issued by the generic revised simplex code is small compared to the number of operations issues by each backend implementation, hence partially amortizing the cost of these

pointers. In our case, the flexibility to easily develop and test new implementations wins out over the marginal performance benefits of a more tightly integrated implementation structure. For example, the CUBLAS backend was developed in several hours by simple modifications to a copy of the CUDA backend.

In addition to the core revised simplex abstraction and implementation, we also developed code to read the common MPS file format for describing LP problems [2, 3] (see `src/read_mps.c`). MPS allows for sparse representation of constraints and the objective function using a key-value based approach, but also allows for heterogeneous constraint types and several auxiliary specifications as mentioned in section 2. For simplicity, and due to time constraints, we chose to only implement the core features required to read several of the Netlib test problems [4] (see section 4.2). In particular, we left the implementation of the `RANGES` and `BOUNDS` sections of MPS for future work as these sections are no more than different ways of augmenting the main constraints. Therefore they raise no new mathematical or algorithmic issues. Our MPS implementation executes several simple sanity checks on the incoming LPs to eliminate any all-zero rows and columns, to transform equalities into two inequalities, to homogenize the directions of constraints, and to standardize the direction of the objective function. All of the aforementioned pre-processing routines proved useful in reading one or more of the Netlib problems.

4 Results

We seek to understand the performance characteristics of our OpenMP and CUDA implementations of the revised simplex method in comparison with a serial base-line. To this end, we executed as a suite of *microbenchmarks*, which exercise the primitives used by the revised simplex algorithm. These microbenchmarks include data-movement operations, i.e., *get* and *set*, as well as critical linear algebra operations, e.g., matrix inverse, dot product, and matrix multiplication, and the specialized operations described in §3. Then, we present the performance of our implementations on a curated set of LP optimization problems from the Netlib [4] repository.

The three main objectives of our analysis are:

- To measure the performance of the parallel implementations of the operations used in the simplex algorithm (§4.1).
- To measure the performance of our multi-threaded revised simplex algorithm on a well-known set of test problems (§4.2).
- To measure the performance of our GPU revised simplex algorithm against an implementation utilizing the highly optimized cuBLAS library (§4.3)

4.1 Microbenchmarks

As noted in section 3, we chose to implement the revised simplex method in such a way that it can use an arbitrary set of implementations, serial or parallel. Thus, our parallel implementations of the algorithm are made by parallelizing the respective operations required by revised simplex. In this section, we present the performance of these operations in isolation on a custom-built benchmarking suit. Our benchmark suit creates an arbitrary set of input data for the set of operations, and accepts several parameters including the number of iterations in which to run each operation and the input data set size.

Operations: The operations that we target for our microbenchmarks can be divided into memory management routines and compute routines. The set of operations for memory management is:

```
matrix_get_col()
matrix_set_col()
vec_get()
vec_set()
matrix_extract_vec()
vec_extract_vec()
```

Of these `matrix_extract_vec()` returns a new matrix formed with the columns of the old matrix indicated by the given vector. The `vec_extract_vec()` operation has a similar semantic on vectors. These operations are required in both phases of the revised simplex method to extract the current basis matrix $B = A_{\mathcal{B}}$, the residual matrix $A_{\mathcal{R}}$, and the entries of the cost function $c_{\mathcal{B}}$. Note that while we implement more getters and setters as part of our abstract interface, they are not called from the main revised simplex loop and we, therefore, do not judge their performance to be directly relevant.

The compute routines are:

```
vec_min()
vec_subtract()
vec_dot_prod()
vec_matrix_mult()
scalar_vec_mult()
matrix_matrix_mult()
matrix_inverse()
matrix_vec_mult()
matrix_colsum_ltz()
phase1_min_ratio()
phase2_min_ratio()
```

As discussed in §3.1, the last three of these compute routines are combined operations defined specifically to satisfy different steps of the revised simplex algorithm. The other compute routines are common linear algebra operations as the names suggest.

Architecture: We collect the following results while running our code on Talapas, the University of Oregon's supercomputer[1]. The operating system is Red Hat Enterprise Linux, version 7.6. We use one node on the `gpu` partition, with a dual-socket Intel(R) Xeon(R) CPU E5-2690 v4 with with 14 cores per socket, for a total of 28 independent threads running at 2.60 GHz each. The GPU is a Tesla K80, with 24 GB of memory. The programs serial and OpenMP programs are written in C, and the CUDA kernels are written in CUDA/C++. We use the gcc/g++ 7.3.0 and NVCC 9.2 for CUDA/C++.

Evaluation Parameters: For these microbenchmarks, we evaluate our implementations with randomly generated n-vectors and n-square matrices, where n = 500. The values in each cell are double precision floating points between -100 and 100. For each benchmark, we run 10 iterations, where new sets of random numbers are used in each iteration. The number of iterations is limited due to the time required by the serial baseline, which can extend upwards of a few minutes for operations such as matrix inverse. Although the number of iterations is low, the variance in time for each set of benchmarks is also low, as seen by the box an whisker plots.

OpenMP: Figure 1 shows the speedup for the test set of operations between our OpenMP implementation and the serial baseline. We observe the best performance boost for the matrix inverse operation, where the average serial time is 5.5 seconds and the average parallel time is 0.3 seconds, approximately 18 × faster. Other operations, such as `matrix_extract_vec` and `matrix_vec_mult` have more modest performance benefits, below 10 ×. Some operations, including `col_sum_ltz`, and `phase1/phase2_min_ratio` have no performance change because they are not parallelized. We attempted to parallelize these functions, however, the operations required custom reduction operations whose implementation was still giving race conditions at the time of submission. Some operations, such as `matrix_get_col` appear to have a wider variance because the absolute time for these operations was on the order of 10^{-6} s to 10^{-5} s.

Cuda: Figure 2 shows the performance increase for the CUDA implementation of our microbenchmarks. As we can see, we have greatly increased performance with respect to the serial code, with operations such as `matrix_set_col`, `matrix_inverse`, `vec_dot_prod`, `matrix_vec_mult` and `matrix_matrix_mult` all performing 10 to 100 × faster than the serial version. Other random-access memory operations are drastically slower with CUDA. For example, `vec_get` and `vec_set`, which read or write a randomly chosen element in an array, are more than 100 × slower against the serial code. Other Vector operations are also slower with respect to the serial operations such as `scalar_vec_mult`, and `vec_dot_prod`. This is likely because the size of the arrays, 500 elements, is not large enough to show the performance boost in the GPU.

cuBLAS: Our CUDA implementations are admittedly not highly optimized and likely have room for improvement. In order to gauge our code's potential for improvement, we investigate how our code compares with a highly-optimized state-of-the-art linear-programming library, cuBLAS [1]. Figure 3 shows the results using the same microbenchmarks considered in this section. Here, we see that performance gains are pushed even further with cuBLAS, upwards of 100 times faster. Note that, because cuBLAS only supports the common BLAS linear algebra operations, some methods in the cuBLAS backend, such as `phase1/phase2_min_ratio_test`, fall back on our CUDA kernels. Thus the performance of these is, as expected, the same as our CUDA implementation. Surprisingly enough, the vector operations all took a performance hit from the cuBLAS operations as well. However, this is again, likely due to the relatively small size of the vectors. We leave extending these microbenchmarks to a comparison between parallel implementations over larger problems sizes for future work.

4.2 LP Evaluation

To understand the impact of our three different backends on the performance of the revised simplex algorithm, we solve a selected set of LP problems from the Netlib problem set [4]. Our selection of problems is limited by our minimal implementation of the MPS format (see §3.2) and the fact that several of the Netlib problems contain tricky conditions which cause our revised simplex implementation to cycle. Table 1 shows our selection and the characteristics of these problems.

Our serial and OpenMP implementations are able to solve each of the problems listed in table 1. With the exception of the last two problems in the table, we run each solver on each one 30 times. In the case of SHIP04S and SHIP04L, we only run the solver 3

[1] `https://hpcf.uoregon.edu/content/talapas`

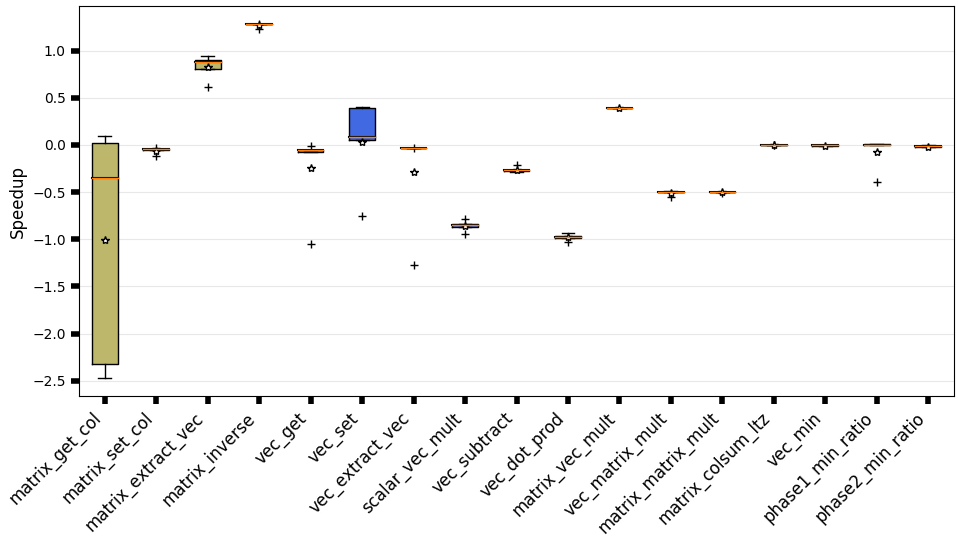

Fig. 1. OpenMP Operation Speedup vs Serial, log_{10} scale

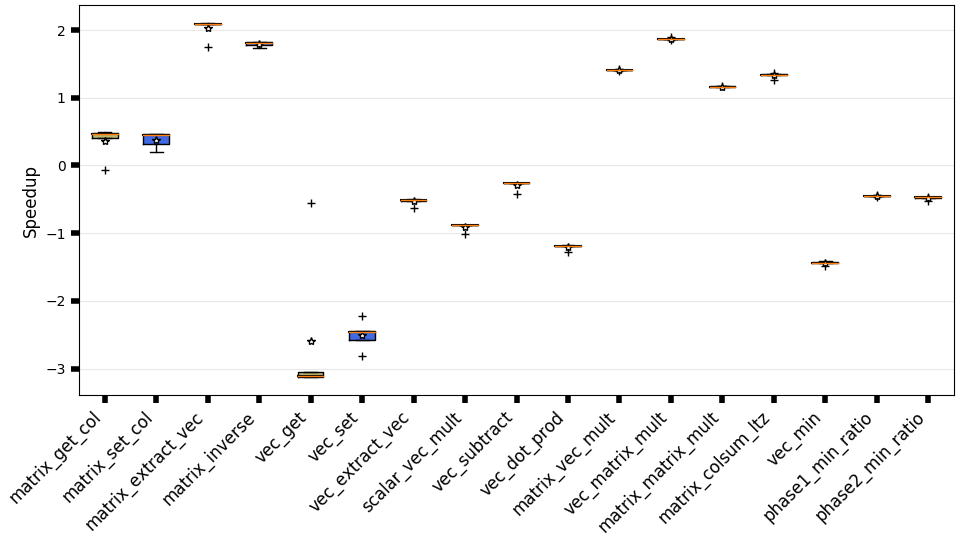

Fig. 2. CUDA Operation Speedup vs Serial. log_{10} scale

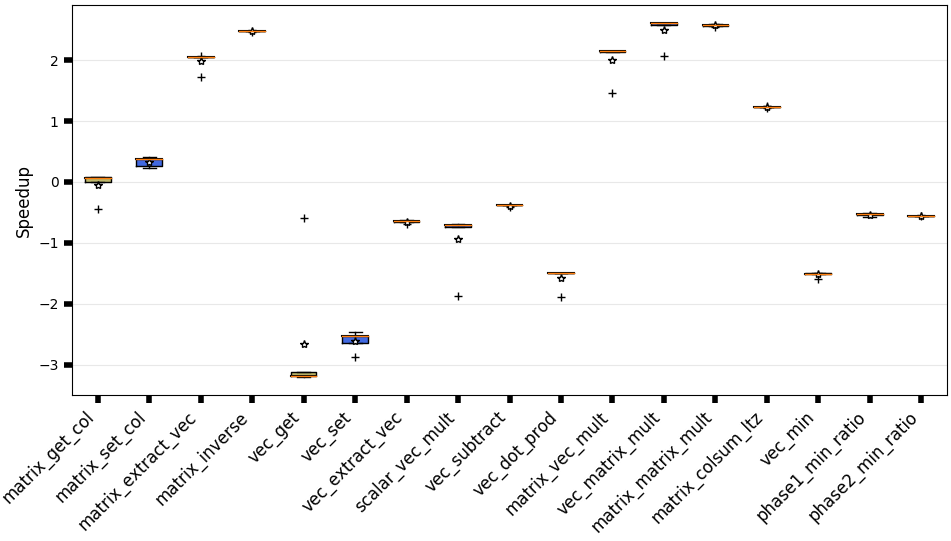

Fig. 3. CuBLAS Operation Speedup vs Serial, log_{10} scale

Name	Rows	Columns	Non-Zeros
ADLITTLE	57	97	465
AFIRO	28	32	88
BLEND	75	83	521
ISRAEL	175	142	2358
LOTFI	154	308	1086
SC105	106	103	281
SC205	206	203	552
SC50A	51	48	131
SHARE2B	97	79	730
SC50B	51	48	119
SCAGR7	130	140	553
SHIP04S	403	1458	5810
SHIP04L	403	2118	8450

Table 1. NETLIB Test Problems and Their Sizes.

times, and only with CUDA and cuBLAS back-ends. This is because even our CUDA back-end took several minutes to complete these problems. A single test of the serial backend on SHIP04S ran for over an hour before terminating. In the following results, we report absolute time and speed up based on the time taken by all iterations of the main revised simplex loop, not accounting for the time spent setting up variables and moving data. This decision is motivated in part by our original impetus to employ this LP optimization as part of a larger IP optimization, in which case, the data would already be in place.

Figure 4 shows the absolute time taken to complete each of the linear programs for our serial, OpenMP, and CUDA backends. In this figure, we see that for small problems, the serial implementation outperforms both parallel implementations. However, as the size of the problem increases, the parallel algorithms increase in efficiency and are able to outpace the serial version, which grows dramatically in runtime. This is consistent with our prior experience using off-the-shelf LP optimization tools which may take hours to solve problems beyond a certain size.

To further illustrate the effect of problem size, figure 5 shows the speedup of our OpenMP and CUDA backends against the serial version. We see that the best speedup from CUDA is on SCAGR7, a problem with 130 constraints and 140 variables, where it gives a solution 3.5 $\times$ faster than the serial version. The best problem for OpenMP is SC205, with 206 constraints in 203 variables, where it performs $\sim$ 2.2 $\times$ faster than serial, and surprisingly also outperforms CUDA. These results demonstrate that, while problem size is a good indicator for the potential for parallel implementations to achieve speed up, it is not the only indicator—other factors, such as numeric precision and memory latency, also play a key role in the revised simplex method's ability to arrive quickly at the optimal solution.

4.3 Comparison with cuBLAS

After evaluating how our implementations perform with respect to each other, we step beyond and see how our CUDA kernels compare with the state-of-the-art cuBLAS linear algebra library, when aimed

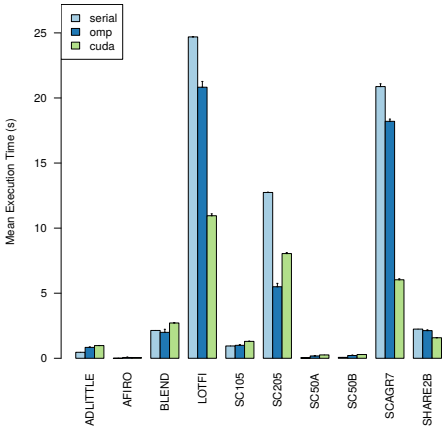

Fig. 4. Execution time for 10 NetLib problems

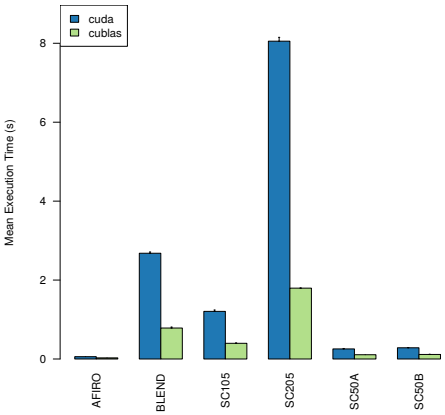

Fig. 6. Execution Time Comparison for Our CUDA back-end vs cuBLAS

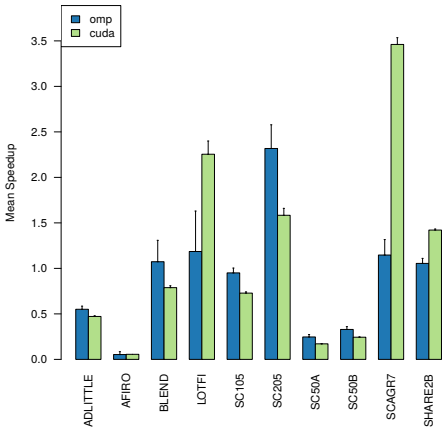

Fig. 5. Speedup for 10 NetLib problems

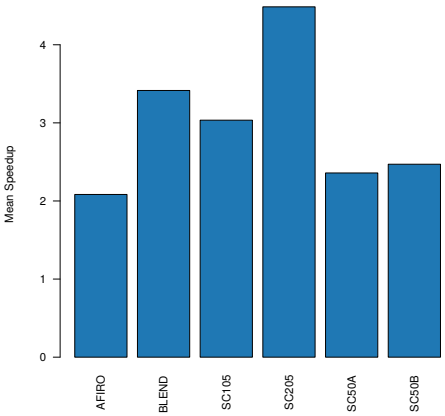

Fig. 7. Speedup for Our CUDA back-end vs cuBLAS

at solving LP problems. Figure 6 shows the time taken to complete several of the LPs from table 1. We see that for larger problems, the performance gap between our CUDA implementation, and the cuBLAS kernels grows. Figure 7 shows the speedup for cuBLAS. We can see that cuBLAS is generally 2 to 5 × faster. For the two largest problems the gap was more pronounced. Our CUDA back-end finishes SHIP04S in ∼212 s, while cuBLAS completes in only 13 s, giving a speedup of more than 16 ×. For SHIPO4L, the speedup is ∼ 19 ×. While we do not have thorough knowledge of the particular optimizations leveraged in the cuBLAS library, we suspect these differences are caused in part by better use of the GPU cache hierarchy and better thread block and grid dimensioning. Additionally, the cuBLAS uses an LU-decomposition method to implement the matrix inverse operation while our CUDA implementation uses a parallelized form of Gauss-Jordan elimination.

It should be noted that several LPs are not shown in figures 6 and 7. Among those missing from the cuBLAS evaluation are ADLITTLE, BLEND, LOTFI, and SHARE2B. This is a result of the fact that the cuBLAS back-end was unable to solve these problems with out cycling, in some cases before an optimal solution was reached. Due to the stronger optimizations employed in cuBLAS and the alternative matrix inversion method, we suspect this issue is a result of numeric instability. This observation is supported by the fact that these same problems were troublesome in our early tests of the original serial backend and required rounding near-zero values to zero in the matrix inversion operations and elsewhere.

5 Related Work

The utility of linear programming for solving complex optimization problems across disciplines has lead to a considerable development of theoretical and practical literature on the topic. Significant effort has gone towards developing efficient sequential algorithms under various constraints [8, 12, 23], as well as parallel algorithms [5,22,25] and GPU-based implementations [6,13,28].

The OpenMP work in this paper is related most closely to that of Ploskas *et al.* [24], who present an OpenMP implementation of the revised simplex algorithm. In their work, they focus on achieving speedup via more efficient basis inversion techniques. Among those are the product form of the inverse (PFI), and Modification of the Product From of the Inverse (MPFI). They show a speedup of 1.79 to 1.44 over the serial version using PFI or MPFI respectively.

The work of Gahrousei *et al.* [13] also implements the revised simplex method on a GPU system. This effort evaluates their implementation on randomly-generated dense problems as well as Netlib. They claim a speed up of 25 × for randomly generated problems, and 65 × for the Netlib problems.

The revised simplex method studied here is also closely related to Integer and Mixed-Integer Linear Programming (MILP) optimization. MILP is a more computationally expensive process, which runs simplex multiple times in order to find integer solutions when such constraints are given in a problem formulation. MILP was first addressed in the literature through methods based on linear programming [16] but more recently tree-search methods have proven to be more performant and precise [10,19,21,25]. Of these search methods, the branch and bound procedure has developed a large body of parallel optimizations and several GPU-specific implementations are documented [15,30].

6 Conclusion

In this paper, we described our parallel implementations of the revised simplex method for solving Linear Programming optimization problems. Our methodology was to write a set of core operations required by revised simplex, and then write a C program with an interface to use any arbitrary backend set of operations, be they single-threaded, multicore, or GPU. We wrote OpenMP and CUDA backends for the operators and evaluated their performance against the serial code. We also presented microbenchmarks for the performance of these individual operations on different architectures. In addition to our OpenMP and CUDA implementations, we also evaluated these microbenchmarks for cuBLAS, the state of the art linear-algebra backend for NVIDIA GPUs. We found that, our CUDA and OpenMP implementations were capable of outperforming all but the smallest problems that we evaluated from Netlib, and that our OpenMP and CUDA implementations of the revised simplex were approximately 2 to 4 × faster than the serial version given large enough problem sizes.

7 Acknowledgements

We would like to thank Prof. Choi for an engaging and informative term this fall. Also, this work benefited from access to the University of Oregon high performance computer, Talapas.

References

1. Nvidia cublas. `https://developer.nvidia.com/cublas`.
2. Documentation - gurobi. `https://www.gurobi.com/documentation/8.1/refman/mps_format.html`, 2019.
3. Mps file format. `http://lpsolve.sourceforge.net/5.0/mps-format.htm`, 2019.
4. Netlib repository. `https://www.netlib.org/`, 2019.
5. Noga Alon and Nimrod Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 574–582. IEEE, 1990.
6. Jakob Bieling, Patrick Peschlow, and Peter Martini. An efficient gpu implementation of the revised simplex method. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.
7. Karl Heinz Borgwardt. *The Simplex Method: a probabilistic analysis*, volume 1. Springer Science & Business Media, 2012.
8. Kenneth L Clarkson. Las vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM (JACM)*, 42(2):488–499, 1995.
9. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
10. Robert J Dakin. A tree-search algorithm for mixed integer programming problems. *The computer journal*, 8(3):250–255, 1965.
11. George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.
12. Martin E Dyer and Alan M Frieze. A randomized algorithm for fixed-dimensional linear programming. *Mathematical Programming*, 44(1-3):203–212, 1989.
13. Arash Raeisi Gahrouei and Mehdi Ghatee. Effective implementation of gpu-based revised simplex algorithm applying new memory management and cycle avoidance strategies. *arXiv preprint arXiv:1803.04378*, 2018.
14. John C Gardner, Ronald J Huefner, and Vahid Lotfi. A multiperiod audit staff planning model using multiple objectives: Development and evaluation. *Decision Sciences*, 21(1):154–170, 1990.
15. Jan Gmys, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. A gpu-based branch-and-bound algorithm using integer–vector–matrix data structure. *Parallel Computing*, 59:119–139, 2016.
16. Ralph E Gomory et al. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical society*, 64(5):275–278, 1958.
17. Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26. ACM, 2013.
18. Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.
19. Ellis L Johnson, George L Nemhauser, and Martin WP Savelsbergh. Progress in linear programming-based algorithms for integer programming: An exposition. *Informs journal on computing*, 12(1):2–23, 2000.
20. Nasiruddin Khan, Syed Inayatullah, Muhammad Imtiaz, and Fozia Hanif Khan. New artificial-free phase 1 simplex method. *arXiv preprint arXiv:1304.2107*, 2013.
21. Jeff T Linderoth and Martin WP Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.
22. Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming. In *STOC*, volume 93, pages 448–457, 1993.
23. Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM (JACM)*, 31(1):114–127, 1984.
24. Nikolaos Ploskas, Nikolaos Samaras, and Konstantinos Margaritis. A parallel implementation of the revised simplex algorithm using openmp: some preliminary results. In *Optimization Theory, Decision Making, and Operations Research Applications*, pages 163–175. Springer, 2013.
25. Ted Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear optimization. In *Handbook of parallel constraint reasoning*, pages 283–336. Springer, 2018.
26. Ron Shamir. The efficiency of the simplex method: A survey. *Management Science*, 33(3):301–334, 1987.
27. Steve Smale. On the average number of steps of the simplex method of linear programming. *Mathematical programming*, 27(3):241–262, 1983.
28. Daniele G Spampinato and Anne C Elstery. Linear optimization on modern gpus. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.
29. Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, 2004.
30. Trong-Tuan Vu and Bilel Derbel. Parallel branch-and-bound in multi-core multi-cpu multi-gpu heterogeneous environments. *Future Generation Computer Systems*, 56:95–109, 2016.