GPU ACCELERATED SIMPLEX SOLVER

Alain Tissier, Carmen Engele, Dominic Rinderer, Pragnya Rachakonda, Yannik Wyss

ETH Zürich
Zürich, Switzerland

ABSTRACT

Efficiently solving linear programs (LPs) is fundamental to high-performance computing, yet traditional simplex solvers remain predominantly CPU bound. We investigate whether the revised simplex method can be effectively accelerated by the parallel throughput of modern GPUs.

Our implementation utilizes custom CUDA kernels and the Sherman-Morrison formula for rank-one basis updates, validated against the Netlib benchmark suite. We successfully overcame the "overhead wall" typical for iterative GPU algorithms by leveraging CUDA Graphs, which eliminated kernel launch latency and allowed for full GPU saturation. However, we find that for the small-to-medium sparse instances in the Netlib suite, the inherent sequential nature of the simplex method limits performance compared to CPU solvers with large caches.

By identifying these specific bottlenecks and providing a validated conversion pipeline for sparse MPS formats, we characterize the fundamental trade-offs between sequential pivot logic and GPU parallelism. Our findings suggest that future performance gains on GPUs will not come from further kernel optimization, but from adopting computationally intensive pivot strategies such as steepest-edge pricing on massive-scale problems to trade raw compute throughput for a reduction in total iterations.

1. INTRODUCTION

The simplex method is one of the most widely used algorithms for solving linear programs and remains highly effective on large, sparse, real-world instances [1, 2]. Despite the existence of polynomial-time alternatives, simplex continues to dominate practical LP solving due to its robustness, warm-start capabilities and strong empirical performance [3, 4, 5]. As a result, it plays a central role in many real-world optimization problems, including logistics and supply-chain planning, energy generation and transmission scheduling, financial portfolio optimization, traffic assignment and network-flow optimization [6, 5, 7, 2]

Motivation State-of-the-art simplex solvers such as HiGHS, Clp, Gurobi, and CPLEX achieve impressive performance through decades of CPU-centric optimization. However, these solvers rely almost entirely on CPU architectures [8]. In contrast, modern GPUs offer substantially higher parallel throughput and memory bandwidth [9], motivating the question of whether the revised simplex method [10] can be adapted to exploit GPU hardware effectively. If successful, such an adaptation could enable faster solution times for large-scale optimization problems arising in practice [11, 12]. Adapting simplex to GPUs is fundamentally challenging [13, 14]. The algorithm relies on sequential pivot decisions, fine-grained updates, and irregular memory access patterns [13], all of which conflict with the massively parallel execution model of GPUs [14]. Determining which components of the revised simplex method can benefit from GPU acceleration, and which represent inherent scalability limits, is therefore essential.

Related Work Prior research on GPU-accelerated optimization has largely focused on interior-point methods [15, 16, 17], whose dense linear algebra operations map naturally to GPU architectures. Conversely, simplex-based methods are more difficult to parallelize [13]. Existing GPU simplex approaches typically accelerate isolated subroutines [9] rather than providing complete solvers, and their effectiveness on sparse, real-world instances remains limited [14, 13].

To address these limitations, we present a GPU-accelerated revised simplex solver where the entire iteration loop resides on the device. We implement the algorithm using custom CUDA kernels alongside `cuBLAS` and `cuSolver` [18], employing Sherman-Morrison updates for the basis inverse. To enable robust benchmarking, we construct a validation pipeline to convert Netlib MPS files into Compressed Sparse Column format [19, 20]. Our results demonstrate that while we successfully eliminate the architectural overhead of kernel launching using CUDA Graphs [18], the algorithmic requirement for sequential pivots remains the fundamental bottleneck for small-to-medium sparse instances.

2. BACKGROUND

In this section, we summarize the theoretical and computational concepts relevant to our GPU-based simplex solver. We begin with the standard LP formulation, then describe

the revised simplex algorithm and its computational requirements, followed by challenges in sparse linear algebra and benchmarking.

2.1. Linear Programming

A linear program in standard form is written as

$$\min_{x \in \mathbb{R}^n} \; c^\top x \quad \text{s.t.} \quad Ax = b, \quad x \geq 0,$$

where $A \in \mathbb{R}^{m \times n}$ is typically sparse for large-scale problems. Efficient LP solving requires exploiting sparsity while maintaining numerical stability in the presence of degeneracy and ill-conditioning.

2.2. The Revised Simplex Algorithm

The revised simplex method iterates over bases defined by subsets of the columns of A. In each iteration, reduced costs are computed to select an entering variable, then a ratio test is performed to identify the leaving variable, subsequently a linear system of the form $Bd = a_{\text{enter}}$, where B is the current basis matrix, is solved, and finally the basis is updated and the process is repeated.

Rather than maintaining a full tableau, the revised simplex method stores and updates the basis implicitly. This significantly reduces memory requirements but shifts computational cost toward repeated linear solves and factorization updates.

2.3. Sparse Input

Real LP program specifications are typically communicated in a sparse format. We implement our own storage format, that is simpler than the more general formats. In this format the vectors c, b are stored in a dense format, while the matrix A is stored in a CSC format [21]. For the constraints of the variables themselves we assume $0 \leq x$.

2.4. Computational Challenges

The performance of the revised simplex method depends heavily on efficient solves with the basis matrix and on maintaining sparse structures across iterations [22, 23]. While sparse LU factorization is the classical approach for maintaining basis factorizations [23], it often introduces *fill-in* [20] and irregular memory-access patterns [24] that are poorly suited to GPU architectures, motivating the exploration of alternative solver strategies such as iterative methods [22] or the use of specialized GPU sparse libraries [25].

2.5. Benchmarking with Netlib

The Netlib LP test set [26] provides sparse, real-world LP instances with diverse structures and numerical difficulties. Since Netlib uses the MPS file format, we developed a format conversion workflow to translate MPS files into our internal format 2.3. A validation pipeline which uses HiGHS [27] ensures that converted instances preserve the solution of the original problem.

3. PROPOSED METHOD

This section describes the methodology used to build and analyze a GPU-based revised simplex solver. Our approach proceeds through several stages: developing a dense reference implementation, porting the algorithm to the GPU, exploring sparse computational strategies, constructing a benchmarking pipeline, and profiling performance.

3.1. Data Preparation and Presolving

Real-world linear programming problems, such as those in the Netlib test set, often contain redundant constraints and fixed variables that can be eliminated before solving [28]. Since this work focuses on accelerating the revised simplex method rather than on presolving heuristics, we apply a standard presolve step to isolate the core numerical difficulty.

We use the HiGHS presolver [27] to reduce the original problems. The resulting presolved instances were converted to our internal compressed sparse column representation (described in Section 2.3) and used as input to the GPU solver. This ensures that performance measurements reflect the speed of the GPU-accelerated simplex method rather than the handling of trivial redundancies.

Validation. For each problem, we verify correctness by comparing the optimal objective value of the original problem (solved with HiGHS) to that obtained from our solver on the presolved instances. Results are accepted if the values agree within a tolerance of 10^{-6}.

3.2. Implementation Variants

To isolate the impact of architectural optimizations, we developed the solver in four iterative stages:

CPU Reference (Validation). We first implement a dense revised simplex solver on the CPU following standard lecture materials [29]. This serves purely as a baseline to verify numerical correctness and debug pivot rules before introducing GPU complexity.

GPU-naive: Naive Port. This version translates the CPU reference directly to the GPU.

- **Dense Factorization:** It performs a full LU factorization ($O(m^3)$) using `cuSolver` [18] at every iteration.
- **Host-Device Ping-Pong:** While matrix-vector products and LSE solves run on the device, reduced costs and di-

rection vectors are copied back to the host for the ratio test, resulting in high PCIe latency.

GPU-RS: Data Residency & Rank-One Updates. Version 1 targets memory bandwidth and algorithmic complexity.

- **Full Residency:** All primary data structures (A, c, b, basis indices) remain in GPU memory. Pivot selection is ported to custom kernels to eliminate per-iteration PCIe transfers.

- **Sherman-Morrison:** We replace the expensive LU factorization with an explicit basis inverse maintained via Sherman-Morrison rank-one updates [30] ($O(m^2)$). Full recomputation of the inverse is only triggered periodically to correct numerical drift.

GPU-RS+G: Latency Hiding & Kernel optimization. Version 2 targets the "overhead wall" and memory bottlenecks identified in GPU-RS.

- **CUDA Graphs:** We capture the iteration sequence into a CUDA Graph [18], launching batches of iterations asynchronously. This eliminates driver overhead and significantly reduces GPU idle time between iterations.

- **Incremental Updates:** We replace the explicit primal solve (recalculating $x = B^{-1}b$ at every step) with an incremental update ($x_{new} = x - \theta d$). This vector operation is fused with the basis index updates into a single kernel, reducing global memory traffic and launch overhead.

- **Explicit Transpose for Pricing:** We pre-compute and store the transpose of the constraint matrix A^T. This allows the reduced cost calculation ($s = c - A^T y$) to utilize coalesced global memory reads (via `CUBLAS_OP_N`) instead of the inefficient strided access required by on-the-fly transposition.

- **Parallelized Ratio Test:** The GPU-RS implementation relied on a single thread block to synchronize the search for the minimum σ, limiting execution to a single SM and severely underutilizing the device. GPU-RS+G replaces this with a multi-stage parallel reduction that distributes the ratio test across the GPU grid and fuses ratio check and pivot selection into a single memory pass.

- **Optimized Basis Update:** The Sherman-Morrison update was decoupled into a "snapshot" normalization phase and a matrix update phase. This eliminates cache thrashing by separating the pivot row reads from the global matrix writes.

3.3. Robustness

Numerical robustness is a central challenge in practical simplex implementations, especially on GPUs where floating-point behavior, reduced precision, and massive parallelism can amplify instability. Degeneracy, stalling, and near-singular bases are common in real-world LPs and must be handled carefully to ensure convergence.

We mitigate these challenges by applying perturbations and conservative feasibility as well as optimality tolerances, while implementing a Harris-type ratio test [31] and periodically recomputing the basis inverse, following standard practice in revised simplex solvers [22].

3.4. Sherman–Morrison Basis Updates

Instead of relying on LU factorization and sequential triangular solves to compute d and y [29], we maintain an explicit basis inverse to exploit the GPUs dense GEMV throughput. Because pivots constitute rank-one modifications, we avoid the $O(m^3)$ cost of recomputing the inverse at every step by applying Sherman-Morrison updates $O(m^2)$. Given a pivot column a and the computed primal direction $d = B^{-1}a$, the update operation for every element (i, j) is:

$$(B^{-1}_{\text{new}})_{ij} = (B^{-1})_{ij} - d_i \cdot \frac{(B^{-1})_{rj}}{d_r}$$

where r is the index of the leaving row. While analytically efficient, a naive parallel implementation exposes a critical Read-After-Write (RAW) hazard.

In a monolithic kernel, threads updating the matrix must concurrently read the pivot row $(B^{-1})_{rj}$. However, threads assigned to row r are simultaneously writing to these addresses. This contention invalidates cache lines, forcing serialized fetches from global memory ("cache thrashing") and stalling the GPU.

To resolve this, we implemented a decoupled update pipeline (described in Section 3.2):

1. *Snapshot:* A lightweight kernel extracts and normalizes the pivot row into a static, read-only scratch pad.
2. *Update:* The main kernel streams over the matrix, reading the pivot scalars from the scratch pad. This ensures memory coalescing and allows the values to persist in the cache.

Profiling confirms the efficiency of this approach: the optimized kernel achieves a Memory Throughput of 90.5%, indicating that we successfully saturate the VRAM bandwidth and are no longer latency-bound.

3.5. Sparse Wrapper

Our solver approaches the problem in 2 phases. In the first phase we solve problem P' which returns a feasible basis B

that is then used to solve another problem P''. P'' is equivalent to the original problem P and therefore gives a solution for P. The sparse wrapper improves the first phase and the general run time.

Phase 1. In Phase 1, slack variables are introduced to construct a full-rank initial basis, as the algorithm cannot proceed with a singular basis matrix. The objective is to eliminate these slack variables and obtain a feasible basis composed solely of original variables. While this was a costly check in the context of a dense matrix, it has become very simple to check with a sparse representation.

General. The Netlib LP problems are known to be sparse ($\text{nnz}(A) \ll \text{rows} \cdot \text{columns}$). Using a non-sparse data format results in significant performance loss due to moving and copying zero entries. By using a sparse format, namely our internal compressed sparse column representation (described in Section 2.3), we reduce the copy overhead by only keeping non-zero entries.

3.6. Parallelized Harris Ratio Test

We implement the Harris ratio test [31], which requires a two-pass search: first identifying the minimum step length θ_{min}, and second selecting the largest pivot element among candidates within a tolerance of θ_{min}.

A naive GPU implementation (used in GPU-RS) relied on a single thread block to synchronize the search for θ_{min}, leaving the device underutilized. In GPU-RS+G, we implement a two-stage parallel reduction:

1. *Block-Level Reduction:* Each thread block computes a local candidate pair (θ, pivot) using `cub::BlockReduce` [18] which allows the solver to scan the ratio vector d in parallel, utilizing significantly more of the GPUs available resources.
2. *Global Consensus:* A lightweight second kernel aggregates block results to determine the global winner.

This approach distributes the search logic across the entire GPU grid, allowing the ratio test to achieve significantly higher parallel throughput, while maintaining the numerical robustness required for the Netlib test set.

3.7. Latency Hiding with CUDA Graphs

The sequential nature of the revised simplex method creates a significant bottleneck: while our custom kernels execute in microseconds, the CPU-side overhead for launching them dominates the runtime. This "overhead wall" leaves the GPU idle between steps. To overcome this, we utilized CUDA Graphs [18] to capture a *single* iteration's kernel sequence (Pricing, Harris Ratio Test, Basis Update) into a static execution node.

Instead of synchronizing after every step, we issue $k = 100$ asynchronous launches of this graph to the stream in a single batch. This decouples the CPU from the GPU, allowing the driver to queue work faster than the GPU processes it, effectively transforming the solver from latency-bound to throughput-bound. Figure 1 demonstrates a 2.3x speedup for 100 iterations (median 9.04 ms to 3.94 ms) on the `woodw` instance, a representative medium-scale problem where removing driver overhead significantly impacts total runtime.

Speculative Execution & Convergence. Since we do not synchronize with the host during a batch, we cannot check for convergence after each iteration. To handle this, we implement a device-side `optimality_flag`. If a kernel detects optimality (or unboundedness), it sets this flag, causing all subsequent kernels in the remaining graph launches to return immediately (becoming no-ops). While standard libraries like `cuBLAS` which we use to compute the dual and the primal step, lack this predication and continue to execute "wasted" work until the batch ends, this speculative overhead is significantly cheaper than paying the latency cost of a CPU-GPU synchronization at every step.

Batch Tuning. The batch size k represents a tunable trade-off. A larger k amortizes overhead more effectively but increases the latency of detecting convergence and the amount of potential speculative waste. We empirically selected $k = 100$ as a robust balance for the Netlib suite.

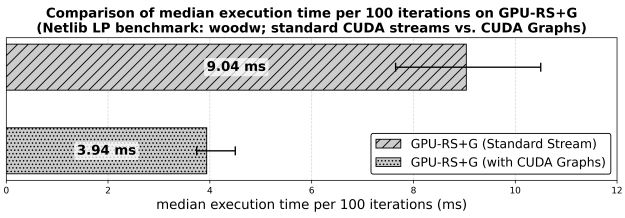

Fig. 1. Performance comparison on the Netlib WOODW problem (batch size 100). Eliminating driver overhead yields a 2.3x speedup (9.04 ms to 3.94 ms). Whiskers denote min/max runtimes.

3.8. Exploration of Sparse Linear Solvers

While the problems we operate on only contain a small number of nnz elements, our algorithm does not reflect this. Most dramatically the operations with Basis B^{-1} contain m^2 elements to operate on. For this reason, we explored options to solve those linear problems $Bx = b$ with different methods that do not require that much data.

For this, we went through a few versions of iterative solvers that tried to minimize the function:

$$f(x) = ||Bx - b||_2^2$$

The initial simple Gradient Descent suffered from divergence, which was solved by the second iteration, where we

chose the step to be optimal. This improved stability but made the convergence oscillate a lot more. To solve this, we then looked into acceleration and optimal acceleration [32], but this still kept iteration numbers high. Our last approach was to work with Normalizer matrices, but the naive preconditioners were ineffective and the complex solutions were too big for this project.

Instead,we looked at the `cuDSS` [33] library from Nvidia. It struggled with multiple features that were advertised by the API but not implemented, such as CSR with start and end vectors and the ability to set the tolerance. With those issues still unresolved, we decided to abandon this exploration and invest our resources in other directions.

3.9. Multi Pivot

The Simplex algorithm itself is very serial, which is why we considered changing this to better utilize the GPUs capabilities.

Instead of doing a single pivot at a time, we explored doing many pivots, which means that we could launch kernels in parallel. Since single-pivot operations leave the GPU underutilized, this increased computational effort could pay off by reducing the total number of sequential iterations.

There were two problems that we face with this, implementation that together made the further exploration uninteresting. For medium and large size (size of LP problems as in 4.1) LP problems the true cost comes from the matrix multiplications. They take most of the time and actually use most of the GPU resources. This means that doing many pivots in parallel just increased execution time by a factor of t, the number of threads. For Small LP problems, the overhead of synchronization between the threads is higher than the time it took to do the iterations in sequence. This negates any potential performance gains.

4. EXPERIMENTAL RESULTS

This section evaluates the robustness, correctness, and performance of our GPU-accelerated revised simplex solvers. We first validate numerical correctness and robustness against a state-of-the-art CPU baseline (HiGHS) on the Netlib LP benchmark suite, including an analysis of degenerate instances. We then analyze the runtime performance of the different GPU variants (see subsection 3.2) relative to the HiGHS baseline.

4.1. Experimental Setup

We evaluate our GPU-accelerated revised simplex solver variants against the HiGHS simplex solver (v1.12.0, single CPU thread), which serves as a CPU baseline. All experiments were conducted on the D-INFK student cluster at ETH Zurich, equipped with an NVIDIA GeForce RTX 5060 Ti GPU, an Intel Xeon E5-2680 v4 CPU, and 24 GB of system memory. CUDA code was compiled using `nvcc 12.0` with optimization flags `-O3` and `-use_fast_math`, while CPU code was compiled with `gcc 13.3.0` using `-O3 -march=native`.

We evaluate our solver variants on the Netlib LP benchmark collection [26]. Evaluation proceeds in three stages: robustness, correctness, and performance benchmarking. For correctness and robustness evaluation, we execute 20 runs per problem instance and solver variant. Each run is subject to a time limit of 240 s. A solver is deemed to execute successfully on a LP instance if repeatedly over 20 runs terminates within the time limit without crashing and correct if the final objective value differs by at most 10^{-6} from the HiGHS reference solution on all runs.

For performance benchmarking, we perform five warm-up runs followed by ten timed runs per problem instance. Reported run times correspond to the median wall-clock execution time, measured on the host. This end-to-end measurement includes GPU kernel execution, kernel launch overhead, synchronization, and host-side control logic, and therefore reflects the runtime observed by an end user. Observed run-to-run variability is minimal (Figure 2).

Problem instances are grouped by size using the following formula $n := \max\{rows, columns\}$: small ($n \leq 10^3$), medium ($n \leq 10^4$), and large ($n > 10^4$); additionally, we analyze highly degenerate instances, defined as problems where at least 50% of the basic variables in the optimal basis are zero.

4.2. Robustness, Correctness and Degeneracy

To validate correctness and numerical stability, we compared the final objective values produced by all GPU solver variants against the HiGHS baseline on all presolved Netlib LP instances. For every problem that converged consistently over 20 runs within the time limit, the objective value was deemed correct if it did not deviate by more than 10^{-6} from the HiGHS baseline, as summarized in Table 1.

Solver	Solved [%]	Correct [%]	Timeout [%]	Crashed [%]
GPU-native	93.1	92.6	6.9	0
GPU-RS	93.1	96.3	4.6	2.3
GPU-RS+G	95.4	97.3	3.5	1.1

Table 1. Robustness and correctness statistics on the presolved Netlib LP benchmark suite (87 instances [26]) with a 240 s time limit. Correct [%] is reported relative to solved instances and denotes agreement with the HiGHS baseline within 10^{-6}.

Due to space constraints, figure 2 reports runtime results only for a representative subset of Netlib LP instances, selected to cover small, medium and large problems as well as highly degenerate stress cases. All remaining instances

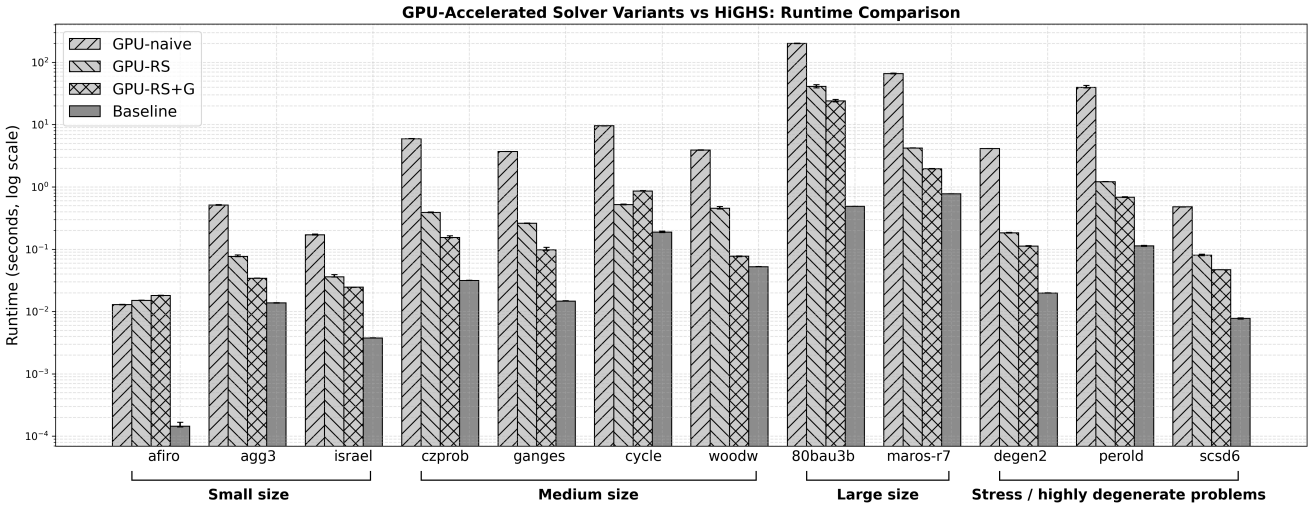

Fig. 2. Runtime comparison of our GPU-accelerated simplex solvers against the HiGHS baseline on a representative subset of Netlib LP problems. Instances are grouped by problem size and degeneracy. Each bar reports the median runtime over ten measured runs following five warm-up runs, with whiskers indicating the minimum and maximum observed runtimes.

exhibited qualitatively similar trends. All non-timed-out instances in the representative subset shown in Figure 2 satisfy the correctness criterion defined above.

4.3. Performance Comparison Against HiGHS

We now focus on runtime performance and compare the efficiency of the different GPU solver variants against the HiGHS baseline. Bar heights in Figure 2 represent median runtimes across repeated runs, while whiskers indicate the minimum and maximum observed runtimes. This is primarily due to the significant overhead of copying data between the host and device during the initialization and transition between Phase 1 and Phase 2, which dominates the runtime for these computationally lightweight instances. As problem size increases, GPU-RS and GPU-RS+G consistently outperform GPU-naive, demonstrating the benefit of data residency, improved kernel fusion and reduced launch overhead.

GPU-RS+G achieves the lowest runtimes on most medium-sized instances, indicating that its optimized basis updates and improved memory access patterns better exploit GPU parallelism. Nevertheless, performance improvements remain limited, reflecting the inherently sequential structure of simplex iterations [11].

For large-scale and highly degenerate problems, performance deteriorates across all GPU variants. Several instances timed-out at 240 s. These cases are dominated by frequent full basis re-inversions to correct numerical drift and synchronization, which significantly reduce effective GPU utilization.

5. CONCLUSION

This work demonstrates that while architectural optimizations can effectively mitigate the latency bottlenecks of the revised simplex method, the fundamental challenge remains the algorithm's sequential nature. By leveraging custom kernels for Sherman-Morrison updates and utilizing CUDA Graphs to mask launch overhead, we developed a GPU solver that eliminates the "overhead wall" and achieves numerical stability comparable to the state-of-the-art HiGHS solver. This proves that reliable LP solving on GPUs is feasible despite floating-point non-associativity.

Outlook. However, our results on the Netlib suite highlight that on small-to-medium sparse instances, the GPU lacks sufficient data parallelism to outperform modern CPU caches. Consequently, future performance gains will not come from further kernel optimization, but from algorithmic shifts. To fully exploit the GPUs massive throughput, solvers must trade raw compute power for a reduction in latency-bound iterations by adopting computationally intensive pivot strategies, such as Steepest Edge pricing [34]. Furthermore, a production-grade implementation must eliminate the remaining PCIe bottlenecks by moving all scaling and phase-transition logic entirely to the device.

Ultimately, while current sparse benchmarks favor CPUs, this foundation paves the way for efficient solving of massive, dense linear programs where GPU parallelism can be fully realized.

6. REFERENCES

[1] James E. Reeb and Scott A. Leavengood, *Using the Simplex Method to Solve Linear Programming Maximization Problems*, Oregon State University Extension Service, Corvallis, OR, em 8720-e edition, October 1998, Operations Research Extension Manual.

[2] Gérard Cornuéjoand Reha Tütüncü, *Optimization Methods in Finance*, Cambridge University Press, 2006.

[3] Haihao Lu, Zedong Peng, and Jinwen Yang, "cupdlpx: A further enhanced gpu-based first-order solver for linear programming," arXiv preprint arXiv:2507.14051, September 2025, arXiv: math.OC/2507.14051.

[4] Robert E. Bixby, "Solving real-world linear programs: A decade and more of progress," *Operations Research*, vol. 50, no. 1, pp. 3–15, January–February 2002, Invited contribution for the 50th anniversary issue of Operations Research.

[5] Kory W. Hedman, Michael C. Ferris, Richard P. O'Neill, Emily Bartholomew Fisher, and Shmuel S. Oren, "Co-optimization of generation unit commitment and transmission switching with n 1 reliability," *IEEE Transactions on Power Systems*, vol. 25, no. 2, pp. 1052–1063, May 2010.

[6] Steve Nadis, "Researchers discover the optimal way to optimize," *Quanta Magazine*, October 2025, Published 13 October 2025.

[7] Hassan Rashidi, "Simulation and evaluation of network simplex algorithm and its extensions for vehicle scheduling problems in ports," *International Journal of Maritime Technology*, vol. 11, pp. 1–12, Winter 2019.

[8] Thorsten Koch, Timo Berthold, Jaap Pedersen, and Charlie Vanaret, "Progress in mathematical programming solvers from 2001 to 2020," *EURO Journal on Computational Optimization*, vol. 10, 2022, arXiv preprint arXiv:2206.09787.

[9] Kalyan Perumalla and Maksudul Alam, "Design considerations for gpu-based mixed integer programming on parallel computing platforms," in *Proceedings of the 2021 ACM Workshop on Parallel and Distributed Algorithms for Decision Sciences (PDADS '21)*, Virtual Event, USA, August 2021, ACM, pp. 1–8, Association for Computing Machinery.

[10] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, 1963.

[11] Daniele Giuseppe Spampinato, "Linear optimization with cuda," Specialization project report TDT4590 project, Norwegian University of Science and Technology, Trondheim, Norway, January 2009, Project for course TDT4590 – Complex Computer Systems.

[12] 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, March 2018.

[13] Nikolaos Ploskas and Nikolaos Samaras, "Efficient gpu-based implementations of simplex type algorithms," *Applied Mathematics and Computation*, vol. 250, pp. 552–570, November 2015, Preprint available at users.uowm.gr.

[14] Qi Huangfu and Julian A. J. Hall, "Parallelizing the dual revised simplex method," *Mathematical Programming Computation*, vol. 10, no. 1, pp. 119–142, March 2018, Earlier version appeared on optimization-online in 2015.

[15] Jin Hyuk Jung and Dianne P. O'Leary, "Implementing an interior point method for linear programs on a cpu–gpu system," *Electronic Transactions on Numerical Analysis*, vol. 28, pp. 174–189, 2008.

[16] Felix Liu, Albin Fredriksson, and Stefano Markidis, "A gpu-accelerated interior point method for radiation therapy optimization," *arXiv preprint arXiv:2405.03584*, 2024.

[17] Haihao Lu and Jinwen Yang, "An overview of gpu-based first-order methods for linear programming and extensions," *arXiv preprint arXiv:2506.02174*, 2025.

[18] NVIDIA Corporation, *CUDA Toolkit Documentation*, 2024, `https://docs.nvidia.com/cuda/`.

[19] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, 2 edition, 2003.

[20] Timothy A. Davis, *Direct Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2006.

[21] Jack Dora, "Compressed column storage (ccs)," now CSC, `https://netlib.org/linalg/html_templates/node92.html`, Accessed: January 2026.

[22] Philip E. Gill, Walter Murray, Michael A. Saunders, and Margaret H. Wright, "Sparse matrix methods in

optimization," *SIAM Journal on Scientific and Statistical Computing*, vol. 5, no. 3, pp. 562–589, September 1984.

[23] GAMS Development Corporation, *GAMS/MINOS User's Guide*, GAMS Development Corporation, Washington, DC, 2010.

[24] Shaoyi Peng and Sheldon X.-D. Tan, "Glu3.0: Fast gpu-based parallel sparse lu factorization for circuit simulation," *arXiv preprint*, August 2019, arXiv:1908.00204.

[25] NVIDIA Corporation, *cuSPARSE Library*, NVIDIA, Santa Clara, CA, 2024, Version 13.1 Documentation.

[26] Netlib Repository, "Linear programming test problems," `https://www.netlib.org/lp/data/index.html`, 1990, Accessed: 2025-12.

[27] Q. Huangfu and J. A. J. Hall, "Highs – high performance software for linear optimization," `https://www.maths.ed.ac.uk/hall/HiGHS/`, 2018, Accessed: December 2025.

[28] Erling D Andersen and Knud D Andersen, "Presolving in linear programming," *Mathematical Programming*, vol. 71, no. 2, pp. 221–245, 1995.

[29] Yin Zhang, "CAAM lecture notes: (revised) simplex method," Lecture notes, CAAM 378, Rice University, n.d., Accessed: 2025-12-31.

[30] J. Sherman and W. J. Morrison, "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix," *Annals of Mathematical Statistics*, vol. 21, no. 1, pp. 124–127, 1950.

[31] P. M. J. Harris, "Pivot selection methods of the devex lp code," *Mathematical Programming*, vol. 5, no. 1, pp. 1–28, 1973.

[32] Rasmus Kyng and Maximilian Probst Gutenberg, "Advanced Graph Algorithms and Optimization," 2025, `https://github.com/mesimon/agao25_script/raw/main/agao25_script.pdf`, Accessed: January 2026.

[33] NVIDIA Corporation, *CUDA Direct Sparse Solver Documentation*, 2024, `https://docs.nvidia.com/cuda/cudss/`, Accessed: January 2026.

[34] Donald Goldfarb and John K Reid, "A practicable steepest-edge simplex algorithm," *Mathematical Programming*, vol. 12, no. 1, pp. 361–371, 1977.

A. ROBUSTNESS

Stalling and Degeneracy. Stalling occurs when successive pivots fail to make progress in the objective value due to degenerate basic variables. This issue is amplified on GPUs due to parallel execution and floating-point effects. We mitigate stalling by applying perturbations and conservative feasibility and optimality tolerances, allowing progress despite near-degenerate pivots.

Harris Ratio Test. Standard ratio tests are prone to cycling on GPUs due to floating-point non-associativity and low-precision noise. To mitigate this, we implement a Harris-type ratio test. Harris' ratio test considers a tolerance-based set of leaving candidates and selects the one with the largest pivot element, reducing the risk of numerically weak pivots; particularly important for Sherman-Morrison updates.

Robust Pricing. Reduced cost computation (pricing) is another source of numerical sensitivity. Small reduced costs can cause cycling or false optimality; we therefore apply conservative thresholds when selecting entering variables, improving robustness at the cost of slightly slower convergence.

LSE-Style Refactorization. While Sherman-Morrison updates are efficient, accumulated error can degrade accuracy; we therefore periodically recompute the basis inverse using LU factorization in an LSE-style refactorization step.

B. EXPLORATION OF SPARSE LINEAR SOLVERS

While the problems we operate on usually only contain a small number of nnz elements our algorithms do not reflect this. Most dramatically the operations with A_B^{-1} contain m^2 elements to operate on. For this reason we explored options to solve those linear problems $A_B x = b$ with different methods that do not require that much data.

For this we went through a few versions of iterative solvers that tried to minimize the function $f(x) = ||A_B x - b||_2^2$:

1. Simple Gradient Descent. For this we used a Gradient Descent step with a fixed λ to move towards the solution. This had 2 problems it used too many iterations (1000+ for $n = 10$ matrices) to converge and sometimes diverged.
2. Dynamic Gradient Descent. Since we deal with a very simple function f in each step we can actually choose the step size to find the minimum along that direction. This means that we do not diverge anymore, but still we took many iterations for small matrices.
3. Accelerated Gradient Descent. Looking into our convergence we discovered that we often overshot and iterated between similar directions. A typical fix for this is Acceleration. Acceleration helped in many cases, but once again showed signs of divergence in some cases.
4. Provable Accelerated Gradient Descent. For this we took inspiration in the Advanced Graph Algorithms and Optimization lecture notes [32]. Implementing this yielded a stable solver, but once again it took to many iterations for a small matrix.
5. Corrected Gradient Descent. The main issue behind long convergence times is the irregularities of the random matrices we use. If $A \approx I$ the convergence would be a lot faster. To fix this we find a simple matrix M with $M \approx A_B^T A_B$ which can easily be inverted and use M^{-1} as a preconditioner. This improved the convergence but still was to larger for small matrices. This exploration got cut short, because any more complex M are simply out of scope for this project.

After looking into the `cuDSS`[33] library by Nvidia we tried to use it, but we struggled with multiple features that were advertised by the API but not implemented, like CSR with start and end vectors and the ability to set the tolerance. With those issues still unresolved, we decided to abandon this project and invest our resources in other directions.

C. MULTI PIVOT

Looking through our solver we notice a potential optimization opportunity, many of the kernels we implement need some sort of synchronization, which is why we decided to implement many of them on a single block. While this made the design easy, it leaves a huge optimization window open. There are 36 SMs on our machine, since a block can only be executed on a single SM this means that more than 97% of our machine is not utilized during those operations.

To better exploit those resources we decide to work towards a higher throughput implementation, maybe instead of doing a single pivot at a time, we can do many pivots, which means that we can launch kernels in parallel. We therefore decide to explore the possibility of searching through many different pivot steps with different threads and synchronizing between them from time to time.

Shared / Private Workspace – The memory footprint of an iteration is significant. There are 2 big data structures that need to be moved in and out of cache and we would like to keep them in cache for as long as possible. This would be the matrix A and the materialized A_B^{-1}. To make sure that we do not have a copy of each of those for every possible implementation we clearly distinguish between the shared structures and the private ones. We try to put as much into the shared part as possible. This works with all big structures trivially except for A_B^{-1}. Since we want to have B_0 the basis of thread 0 and B_1 the basis of thread 1 to diverge, this also means that $A_{B_0}^{-1} \neq A_{B_1}^{-1}$.

To solve this we add a new data structure we call the $SparseUpdate$ into the mix. We upgrade the Sherman-Morrison Basis Update with this data structure, it allows us to remember up to k updates to be applied to an inverse matrix before we have to apply it.

Thread Coordination – Since thread operations come with overheads we decided to have our threads as long living as possible. For this each thread gets spawned within the core solver and only gets joined once the core ends again. To keep the threads in sync, meaning that we want to join them together again after k iterations we use barriers. After the barrier we access the shared memory figure out which thread made the most progress before continuing with the now updated A_B^{-1} from all threads at the same time.

Problems – There are 2 problems that we faced with this implementation, that together made the further exploration non interesting.

1. For medium and large size (size of LP problems as in 4.1) LP problems the true cost came from the matrix multiplications, they take most of the time and actually use most of the GPU resources, this means that doing many problems in parallel just slows us down by a factor of t the number of threads.
2. For Small LP problems the overhead of synchronization between the threads was higher than the time it took to do the iterations in sequence. With this we also lose any gains we hoped to achieve.