# OPTIMIZING SIMPLEX IMPLEMENTATION FOR STANDARD FORM PROBLEMS

*Donjan Rodic, Rico Häuselmann*

Department of Mathematics
ETH Zürich
Zürich, Switzerland

## Todo list

## ABSTRACT

We implement Dantzigs Simplex Algorithm for Linear Programming, a widespread and established linear optimization method. Various computational optimization techniques are used to improve the efficiency of the algorithm and explore the options of improving upon this highly memory-bound problem; and to compare it to well-known industry tools. Our specific focus on the sub-class of feasible standard maximization problems allows us to achieve performance close to the hardware maximum.

## 1. INTRODUCTION

In this section we will motivate our choice of algorithm to optimize and give a brief overview of related work.

**Motivation.** The Simplex algorithm for linear optimization is widely used in logistics, optimization of business performance and many other fields.

The simplex solution to linear problems (LPs) may be used as a crucial part in basic decision making wherever management requirements for interdependent resources produce linear constraints. In production environments performance is often crucial, therefore it is interesting to see how far a solver for a specific class of linear problems can be optimized.

**Implementation.** We present an implementation which is geared towards a limited class of linear optimization problems and solves this kind of problem faster than any commercially available general purpose solver.

**Related work.** Many production grade linear optimization packages exist. We used a number of them to compare the performance of our implementation to what is being used in big business.

- GLPK [1], a free open source linear programming package.
- Gurobi [2], a commercial grade package which claims to be the fastest available solution when taking into account multi-threading. Gurobi is free for educational use.
- SoPlex [3], free for educational use, developed at Zuse Institute Berlin (ZIB).
- CPPLEX [4], a free open source mathematical implementation emphasizing clean object oriented code style.

## 2. BACKGROUND: THE SIMPLEX ALGORITHM

It follows a brief introduction to the class of problem in consideration and the algorithm used as well as a cost analysis of the algorithm.

**Linear Optimization Problems [5].** The general formulation of a linear problem is

$$\begin{aligned} \max c^T x, &\quad \text{Subject To} \\ b_l \leq Ax \leq b_u, & \end{aligned}$$

Where $c \in \mathbb{Q}^n$, $x \in \mathbb{R}^n$, $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{R}^m$.

**Standard form Linear Problems.** A linear problem is considered to be in standard form when it is formulated as

$$\begin{aligned} \max c^T x, &\quad \text{Subject To} \\ Ax \leq b & \\ x \geq 0. & \end{aligned}$$

Any linear problem can be transformed into standard form [6]. Considering this we chose to restrict our solver to standard form problems, since supporting other formulations would have lead to more effort being put into transforming the problem into standard form (effectively implementing a domain specific language).

**The Simplex Method.** The tableau form of the simplex method for standard form problems works as follows: Let $m$ be the number of constraints and $n$ the number of variables.

1. initialize the tableau $T \in \mathbb{R}^{(m+1)\times(n+m+1)}$

$$T = \begin{bmatrix} A & I_m & b \\ c & 0 & 0 \end{bmatrix}$$

And create a set of column indices to keep track of the currently "active" columns (also called the basis). Initialize the basis with the indices

$$B = \{m, \ldots, n + m - 1\}$$

2. iterate:

   (a) (Pivoting) If $\nexists i \,|\, T_{m,i} < 0$ problem is solved. Else:
   $$\text{pcol} := \text{argmin}\,(T_{m,i})$$

   (b) (Pivoting) If $\nexists i \,|\, T_{i,\text{pcol}} > 0$ problem is infeasible. Else: (Basis Exchange)
   $$\text{prow} := \text{argmin}\left(\frac{T_{i,2m}}{T_{i,\text{pcol}}}\right)$$

   (c) perform Gaussian elimination:
   $$p := T_{\text{prow},\text{pcol}}$$
   for $i \in \{0, \ldots, m-1\}$:
   $$T_{i,:} := T_{i,:} - T_{\text{prow},:}\frac{T_{i,\text{pcol}}}{p},\ if\ i \neq \text{prow},$$
   and update the basis
   $$B := B \setminus \{\text{prow}\} \cup \{\text{pcol}\}.$$

**Cost Analysis.** The cost of the overall algorithm depends highly on the coefficients of the problem, since the coefficients determine the number of iterations necessary. A small problem may very well take more iterations to solve than a huge one.

However, the number of operations contained in one iteration only depends on the problem size. Let the number of iterations be $N$, then

$$\#\text{flops} = N \times 2m \times (m + n + 1)$$

As well as the number of doubles read from memory, which bounded from below by

$$\frac{(m+1) \times (m+n+1)}{B} \leq \#\text{loads},$$

Where $B$ is the block size of the L1 cache for very small problems where the entire tableau fits into cache, and bounded from above by

$$\#\text{loads} \leq N \times (3m + n) + (m+1) \times (m+n+1),$$

Assuming every load is a miss.

For the operational intensity this means:

$$\frac{1}{4} \leq I \leq \frac{B \times N}{4}$$

## 3. OUR IMPLEMENTATION

In this section we explain the our baseline implementation as well as all optimizations we made.

**Baseline Implementation.** Our baseline implementation parses an input file containing the coefficients for the objective function and the constraints. It assumes the problem is in standard form and that the number of constraints is even. The coefficients are stored in a two dimensional STL vector of vector representation of the simplex tableau.

The steps performed in each iteration are separated into three functions:

- `pivot_col`
- `pivot_row`
- `base_exchange`

The first two contain the code for finding the pivot element to use in the iteration whereas the third facilitates the Gaussian elimination step.

**Profiling.** Profiling was done with `gprof`, `perf` and `valgrind`. All tools consistently showed that the vast majority of the runtime is spent in the `base_exchange` function, where we focused all subsequent optimization. Further annotated analysis revealed stalls at the memory access functions, which can be alleviated by exploiting instruction level parallelism (ILP) via static scalar assignment where unit stride access was applicable.

**Array.** The Array optimization replaces the vector of vectors implementation of the tableau by a C99 one dimensional contiguous array allocated and aligned using `_mm_malloc` from the Intel intrinsics headers.

All subsequent optimizations were based on this implementation.

**SSA.** The first optimization consisted of unrolling the tightest loop and applying static assignment in order to increase ILP.

**Blocking.** As a next step we added blocking in the loop that runs over the whole tableau inside the basis exchange. Block sizes from $1 \times 1$ to $16 \times 16$ were tried.

The code for different block sizes was generated using `cheetah` [7], a templating engine for python. The following optimizations were introduced by changing the code templates to be able to make use of the best block size.

**Cache control.** Improvement of cache access attempted through intrinsics for non temporal assignment, in order to avoid polluting the cache with updates on the active basis elements and effective cost calculations. We further added prefetching in order to load soon-to-be-used blocks into the L2 cache, as well as prefetching parts of the current block into L1.

**Swapping.** In order to allow the compiler to perform optimization much better, the pivot and the last row of the tableau were swapped before and after entering the main loop of the base exchange step, allowing to get rid of the if statements used to test for the pivot row in the main loop.

This optimization does not change the number of floating point operations performed, however it affects the number of bytes loaded per iteration slightly.

**Vectorization.** In the last phase, manual vectorization using advanced vector extensions (AVX) instructions was performed.

## 4. EXPERIMENTAL RESULTS

We give a short overview of our hardware and software framework, followed by measurements and visualization of the code performance.

**Experimental setup.** All experiments were conducted on a Core i5-3360M (Ivy Bridge) running GNU/Linux and 32 kB L1 cache, 256 kB L2 cache and 3 MB L3 cache, all at 64 bytes line size.

GCC 4.7.3 was used for compilation. The following flags were used: `-O3 -march-native` for automatic vectorization with `-fno-tree-vectorize -fno-tree-slp-vectorize -mno-abm` additionally for non vectorized code. For profiling `-g3 -ggdb -pg` was added. Turboboost has been taken care of.

A random linear problem generator was written to test the implementations. The problem generator is restricted for simplicity's sake to generate problems with guaranteed feasibility by allowing only positive coefficients. The same parameter was used for the number of Variables as well as the number of constraints.

The problem size was varied from 10 to 1000, or up to 4000 where relevant for out-of-cache comparison. Due to the unpredictable count of steps the Simplex algorithm needs to reach a solution, each problem size was averaged over up to 100 samples. Additionally, in order to achieve a defined minimum of RDTSC cycles, small problems were repeatedly calculated as shown in the exercise framework.

The code was instrumentalized to count floating point and memory operations. The measured numbers for both were compared to manually computed values and found to agree.

In order to compare to third party packages, their respective application programming interface (API) was used according to the available documentation. The same technique was used to measure runtime, however instrumentalization was not used. We used Gurobi [2], SoPlex [3], GLPK [1] and CPPLEX [4], with the last one being dropped from testing because of numerical instability for large problems.

**Results.**

Direct comparison with alternative Simplex solvers (see Fig. 1) shows that even our baseline implementation performs significantly faster than any of the alternatives. This

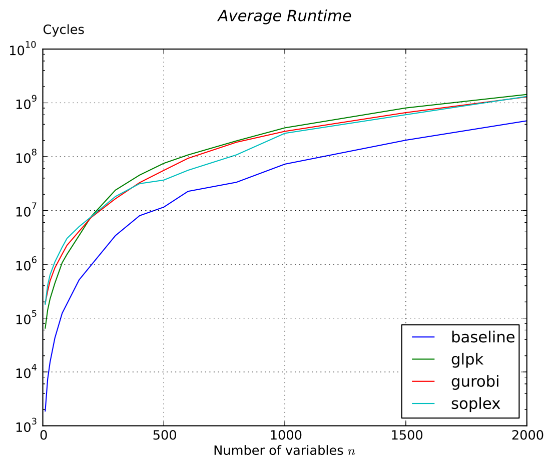

**Fig. 1**. Comparison with Gurobi, SoPlex and GLPK.

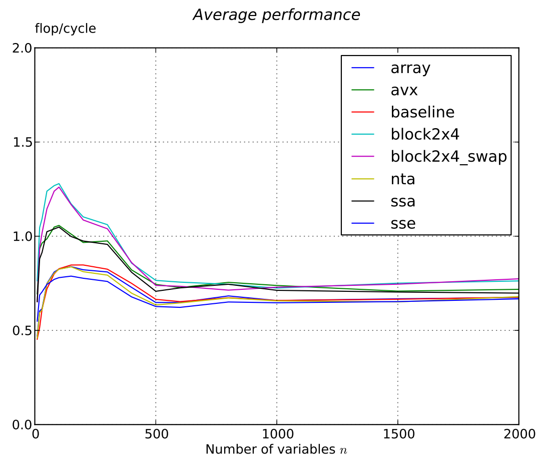

**Fig. 2**. Non-auto-vectorized basic implementations.

results from the simplicity assumptions we have made (regularity, conformity, definiteness) under which a lot of tedious checks can be eliminated. Unfortunately none of the APIs offered to impart this information to the respective solver. We plot the elapsed RDTSC cycles instead of per cycle performance so as not to guess the operation count of each solver.

Of our own optimization techniques, blocking with an additional copy (swap) step showed to be the most efficient for large sizes, both non-auto-vectorized (Fig. 2) and auto-vectorized (Fig. 3). Note that while in theory an AVX enabled program could reach 8 flop/cycle for compute-bound problems on our testing machine, the plots have been capped at 2 flop/cycle for better readability. All further plots contain programs with approximately the same operation count.

Our further goal was to find the optimal block sizes by varying unrolling width and concurrent line elimination

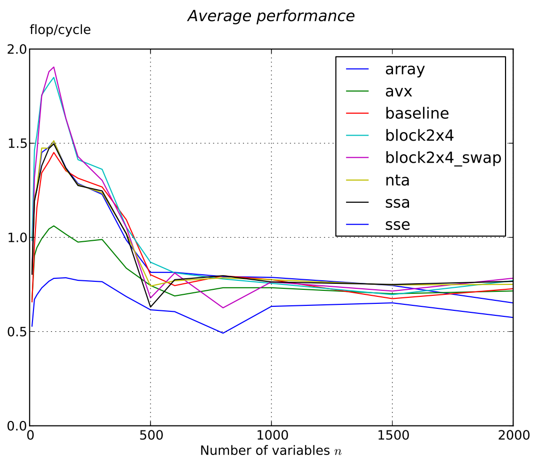

**Fig. 3**. Auto-vectorized basic implementations.

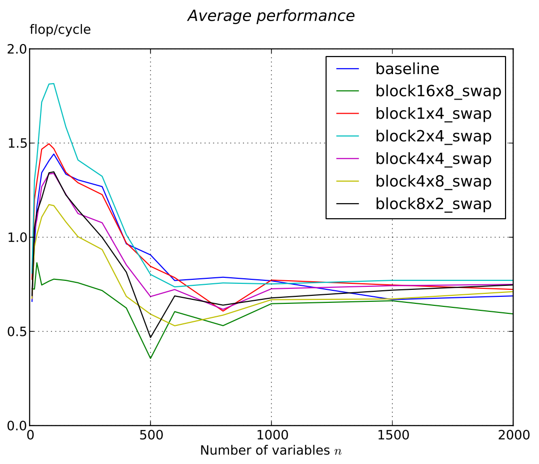

**Fig. 4**. A small selection of the parametrized blocking implementations. Note the kink around $n = 500$, which may indicate a numerically particularly challenging problem.

count. The search was done over incidentally turned out to be our originally picked sizes. The search space consisted of the dyadic product of these two parameters for the dyadic product of AVX and scalar code as well as blocking and non-blocking versions.

For large problems, our most efficient blocking algorithm achieves 0.75 flop/cycle, compared to the baseline 0.68 flop/cycle. Small problems that fit in cache are at the show the greatest difference around $n = 100$ with 1.83 flop/cycle for the blocking algorithm and 1.46 flop/cycle for the baseline. These correspond to speedups of 25% respectively 10%, although we drop to hardly any speedup at the cache boundaries.

Could we improve on this further? As shown in Fig. 5, combining our previous favorite with non temporal assignment resulted in a mostly positive but small effect on the swapping implementations. Any kind of manual prefetching as described in section 3, or partial implementations thereof, lead to a considerable slowdown.

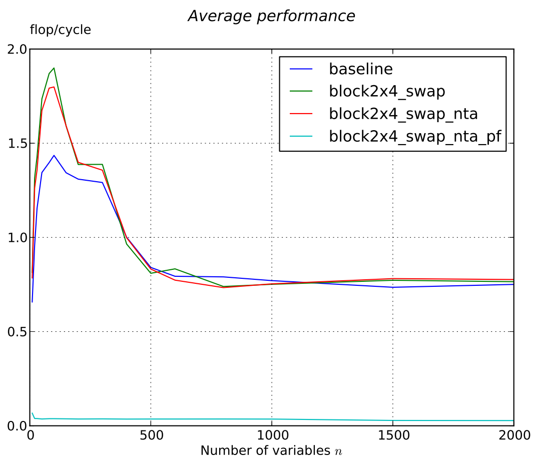

**Fig. 5**. Performance of non temporal assignment (`_nta`) and prefetching (`_nta_pf`) implementations of `block2x4_swap` and the baseline.

One particularly interesting result is that the effect of AVX intrinsics on performance is not easy to predict if we turn on auto-vectorization, yet they do not match the fastest auto-vectorized version (Fig).

Profiling of the `swapNxM_block_avx` implementations showed that between 70% and 90% of the time is spent on `_mm256_load_pd` calls, which once more underlines the memory tightness of the problem.

From the roofline plot shown in Fig. 7 we learn that our best-performing implementation is very close to the theoretical memory limit. The corresponding test set starts with problems of size 500 in order to measure only RAM to CPU traffic.

## 5. CONCLUSIONS

Our measurement results and profiling is highly memory bound and at the same time keeps a very simple structure, which does not lend itself to optimization easily. We were able to realize a speedup of up to 25% for small problems and up to 10% for large ones, albeit limited to feasible standard maximization LPs.

Finally, we can conclude that in cases where many linear problems of only one certain class has to be solved and performance matters it might pay to invest into a specifically tailored solution. Depending on the restrictions of the problem class loss of generality may be rewarded with great speed gains.

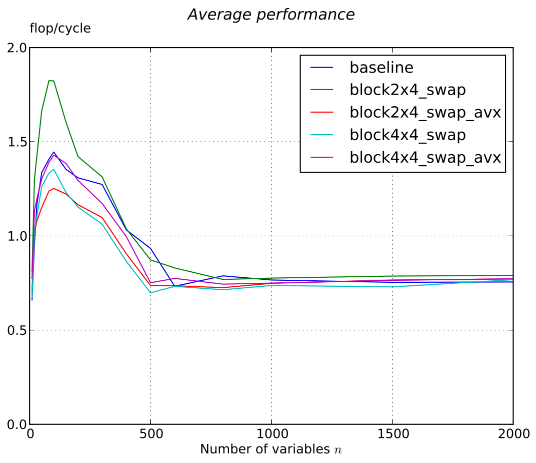

**Fig. 6**. Example comparison of auto-vectorized code and its AVX counterpart.

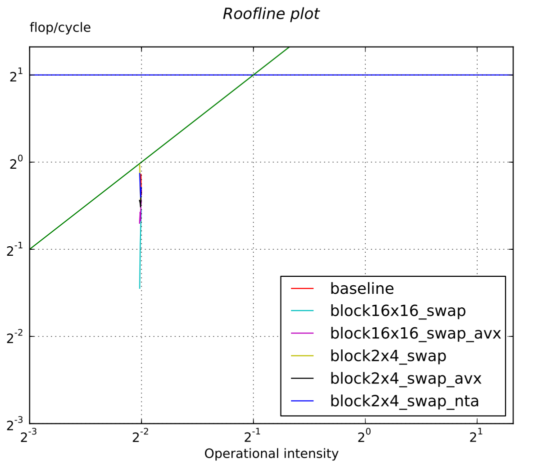

**Fig. 7**. Roofline plot, the 8 flop/cycle performance line is cropped away for better readability.

## 6. REFERENCES

[1] "GLPK (GNU linear programming kit)," 2006.

[2] Gurobi Optimization, Inc, "Gurobi - the overall fastest and best supported solver available," .

[3] "Paralleler und objektorientierter simplex-algorithmus," TR 96-09, ZIB, 1996, http://www.zib.de/Publications/abstracts/TR-96-09/.

[4] Tommaso Urli, "Cpplex - multi-platform, self-contained and object oriented implementation of the simplex algorithm in c++," .

[5] George Dantzig, *Linear Programming and Extensions*, Princeton University Press, 1963.

[6] Martin Grötschel, "Lineare optimierung," Lecture Notes for "Algorithmische Diskrete Mathematik II", 2004.

[7] "Cheetah - the python-powered template engine," 2001.