Exploiting hardware capabilities in interior point methods

Csaba Mészáros

Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest

(Received date to be inserted)

KEY WORDS: Interior point methods, Cholesky factorization, high performance computing, linear programming

The increase of computer performance continues to support the practice of large-scale optimization. Computers with multiple computing cores and vector processing capabilities are now widely available. We investigate how the recently introduced Advanced Vector Instruction (AVX) set on Intel-compatible architectures can be exploited in interior point methods for linear and nonlinear optimization. We focus on data structures and implementation techniques that utilize the new vector instructions. Our numerical experiments demonstrate that the AVX instruction set provides a significant performance boost in our implementation on large-scale problem that have significant fill–in in the sparse Cholesky factorization, achieving up to 100 gigaflops performance on a standard desktop computer on linear optimization problems for which the required Cholesky factorization is relatively dense.

1 Introduction

Interior point methods (IPMs) have proved to be efficient tools in practice for solving large-scale optimization problems [9, 3]. It has also been recognized that the implementations of IPMs are able to exploit a wide variety of hardware features [15]. During the development of IPMs, significant effort was concentrated on exploiting the hierarchical construction of the memory systems of modern computing platforms [20]. When cache memory is exploited efficiently, the performance of IPMs is highly dependent on the speed of the double precision arithmetic computations. The development of floating-point hardware on desktop computers started with the 8087 arithmetic co-processor in 1980, replacing the software emulation that was used before. Starting from the 80486 CPU in 1989 the floating-point unit is integrated into the processor increasing communication and processing efficiency. The development of floating-point hardware continued further and the speed of floating-point instructions became comparable to that of integer instructions in the

Pentium P5 platform, introduced in 1993. Recent improvements in hardware technology were made in the direction of parallel processing by implementing advanced vector instructions and increasing the number of processing cores. The "Streaming SIMD Extensions 2" (SSE2) vector instruction set, introduced in 2001, is able to perform 2 "single instruction multiple data" (SIMD) double precision instructions on 128 bit SSE registers. The subject of our computational study is the performance of the recently introduced AVX vector instruction set in the implementations of IPMs. These new vector instructions use 256 bit registers, allowing 4 SIMD double precision instructions, doubling the throughput of the SSE2 unit. Readers interested in the history of hardware architectures are referred to [24].

In the second section of the paper we introduce the interior point algorithm and the sparse Cholesky factorization, which is the most critical step of implementations. We outline the core implementation techniques used in [21], which is the basis of our further improvements. In the third section some features of the AVX vector processing unit are discussed that are important for efficiency in practice. We describe how the data structures and implementation techniques are adjusted to utilize these features. In Section 4 we present and discuss numerical results on Intel I7-950 (Bloomfield) and I7-2700K (Sandy Bridge) platforms.

2 Interior point methods and the Cholesky factorization

Hereafter we consider the linear programming problem and the primal–dual log barrier interior point method. Note that this choice has only notational consequences because the underlying linear algebra of other interior point approaches is fairly similar.

Let us consider the linear programming problem

$$\begin{array}{rcl} \min\ c^T x & & \\ Ax & = & b, \\ x & \geq & 0, \end{array} \tag{1}$$

where $x, c \in \Re^n$, $A \in \Re^{m \times n}$ is of full row rank, and $b \in \Re^m$. The logarithmic barrier problem corresponding to (1) is

$$\begin{array}{c} \min\ c^T x - \mu \sum_{i=1}^{n} \ln x_i, \\ Ax = b, \quad x > 0, \end{array} \tag{2}$$

where μ is a positive scalar barrier parameter. A log barrier interior point method approaches the optimal solution of (1) by a sequence of barrier problems (2), while the barrier parameter is decreased toward zero. Following the classical introduction of the primal–dual log barrier method, the algorithm can be derived by applying Newton's method to the Karush-Kuhn-Tucker system of (2). The computational

task of the resulting method in every iteration can be reduced to a solution of a system of linear equations whose matrix is ADA^T, where D is a positive definite diagonal *scaling* matrix that changes each iteration [3].

The Cholesky factorization

$$PADA^TP^T = LL^T, \tag{3}$$

is computed, where L is lower triangular and the P is a permutation matrix that preserves sparsity in L. The computed factorization is used in several backsolve operations to determine predictor–corrector [17] and higher–order correction directions [8]. Therefore, the performance of Cholesky factorizations in the implementations is a critical issue [1, 26, 16, 18]. A significant development was done to exploit the sparsity by using fill–reducing orderings [7, 2, 25] that made Cholesky factorizations more efficient. A further significant achievement was to increase the speed of numerical computations by using sophisticated algorithms and data structures that benefit from the processor architectures [20, 21]. One key point of improving the performance of the sparse Cholesky decomposition is using dense computation kernels to exploit the *supernode* structure of the factors [14]. A supernode is a set of continuous columns of L that share the same nonzero structure. Thus, a row in a supernode is either empty or fully dense. The following example shows the supernodal decomposition of a lower triangular matrix with nonzero values marked by a $*$. In the example there are 4 supernodes built from the following sets of columns: (1,2,3), (4,5,6), (7,8,9) and (10,11,12,13,14,15).

$$\left[\begin{array}{ccc|ccc|ccc|cccccc}
* & & & & & & & & & & & & & & \\
* & * & & & & & & & & & & & & & \\
* & * & * & & & & & & & & & & & & \\
 & & & * & & & & & & & & & & & \\
 & & & * & * & & & & & & & & & & \\
 & & & * & * & * & & & & & & & & & \\
 & & & * & * & * & * & & & & & & & & \\
* & * & * & & & & * & * & & & & & & & \\
 & & & * & * & * & * & * & * & & & & & & \\
* & * & * & * & * & * & * & * & * & * & & & & & \\
 & & & & & & & & & * & * & & & & \\
 & & & & & & * & * & * & * & * & * & & & \\
* & * & * & & & & * & * & * & * & * & * & * & & \\
 & & & * & * & * & * & * & * & * & * & * & * & * & \\
* & * & * & & & & * & * & * & * & * & * & * & * & *
\end{array}\right]$$

In our implementation we have adopted the techniques described in [21] for handling supernodes and for parallel computations. Our computational kernel can be described as follows: after the empty rows are removed, supernode B is decomposed into blocks:

$$B = \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ ... \\ B_k \end{bmatrix},$$

where B_0 is a dense symmetric matrix and $B_{i>0}$ is dense rectangular. The supernode partitioning and the decomposition of the supernodes are determined so that the resulting B_i blocks fit entirely into the cache memory. In our factorization, the supernodes are processed one by one. The inner cycle of the factorization computes the numerical values of a supernode and its numerical contribution to the remaining set of the matrix as follows:

1. Compute the Cholesky decomposition $B_0 = L_0 L_0^T$.
2. Do parallel for $i = 1, \ldots, k$

   Update $B_i \longleftarrow B_i L_0^{-T}$ by solving $L_0 \bar{B}_i^T = B_i^T$ and overwriting B_i by $\bar{B}_i$.

   end_do

   for $i = 1, \ldots, k$
3. Do parallel for $j = 1, \ldots, i-1$

   Compute the rectangular update matrix $U_j = B_i B_j^T$.

   end_do
4. Compute the lower triangular part of the symmetric update matrix $U_i = B_i B_i^T$.
5. Do parallel for $j = 1, \ldots, i$

   Subtract the dense update matrices U_j from the remaining submatrix.

   end_do

   end_for

The algorithm exploits the cache memory by re-using the data of L_0 in steps 1 and 2 and the data of B_i in steps 3 and 4. It is easy to see that the operation count is maximal for a given cache size if the blocks $B_1, \ldots, B_{k-1}$ are square. Therefore the partition is determined accordingly.

The last group of columns in L that share the same structure are referred as the *dense window*. The processing of this part of the factorization is especially efficient because the update step can be done in dense mode as well. In our implementation this step is combined with the computation of the update matrices, which further improves efficiency by reducing memory operations.

3 Implementation for the new vector instruction set

The platform on which our numerical experiments are done is Intel's Sandy Bridge architecture, which implements the new set of vector instructions [10]. We outline the most important features and properties of the vector execution unit of this architecture [11] and describe how they are taken into account in our implementation.

The instructions are performed on processing ports that work as pipelines. The floating-point multiplication takes 5 processor cycles to complete and the floating-point addition takes 3 processor cycles. The multiplications and additions are performed on different ports. Therefore a multiplication can be performed in parallel to an addition if there are no data dependencies between these operations. The processing ports can start to process a new instruction in every processor cycle while the previous operations are still processed, i.e., the throughput of these processing ports is one operation per processor cycle. This can be achieved only if there is no data dependency between these operations and the data is available in the processor. In this way, the theoretical maximum of double precision floating-point operation throughput of this architecture is 2 vector operations per processor cycle in every processor core. Since the AVX registers contain 4 double precision data and the standard clock speed of the I7-2700K CPU is 3.9 GHz and the processor has 4 cores, this translates to a theoretical double precision performance of 124.8 gigaflops. This can be achieved with a process that can maintain a sufficiently long queue of independent addition and multiplication instructions. Our other test machine is the older I7-950 CPU that performs SSE4.2 vector instructions in 4 cores at the clockspeed of 3.33 GHz, resulting in 52.8 gigaflops theoretical maximum performance in double precision. In both platforms each core has a dedicated 32K L1 cache and 256K dedicated L2 cache, while the 8MB L3 cache is shared among the cores.

Besides the floating-point computations, any numerical process should supply the necessary input data for the execution units of the processor and store the results. Since memory read/write operations are, in general, slower than the execution of arithmetic operations in registers, this often presents a bottleneck in the computations and degrades performance because of stalls while the execution units wait for the completion of memory operations. The Sandy Bridge architecture has 2 memory read ports that perform 128 bit operations, and since these ports work in parallel as pipelines, the maximal theoretical memory read performance of the platform is 256 bits, i.e., one AVX register per core in each processor cycle. To achieve this performance the data of the vector operations should be in the L1 cache and aligned on 32 bytes boundary. When the data is not aligned or does not reside in the L1 cache, additional penalties apply. Thus, the maximal throughput of one processor core is 2 floating-point vector instructions but only one vector read operation, i.e., each data should be used in at least two arithmetic operations, otherwise the memory operations will be a limiting factor. This means that potentially the matrix–matrix operations can be utilized by the vector instructions efficiently. We investigate steps 3 and 4 of the algorithm, which perform the majority of the floating-point computations of the Cholesky factorization as dense matrix–matrix

multiplications. Note that loading data from the L1 cache memory takes several clock cycles. Thus load operations should be started several clock cycles before the data is needed.

Summing up, an efficient implementation should take into account the following considerations:

- Align data of the vector operations on 32 byte boundary for efficient access.
- Organize cycles on cache line boundary (128 byte).
- Organize the computations into sequences of at least 5 independent multiplications and 3 independent additions.
- Preserve AVX registers to preload data from the L1 cache memory.
- Use prefetch instructions to move data from the higher level cache / main memory into the L1 cache memory.

The previous considerations build a strong case for reducing the data dependency and maximizing the number of floating-point operations with each data loaded into a vector register. Therefore, we decided that the AVX registers will contain column–wise elements from B_i. Since each AVX register stores 4 double precision data, the most natural way is to represent the B_i blocks as sequences of column–wise $4 \times l$ dense submatrices, where l is the number of columns in B. Our process in the inner loop reads one column with 4 double precision elements from B_i and B_j and computes their contribution to the corresponding 16 elements of U_i by performing 4 vector multiplications and 4 vector additions. During these operations an appropriate permutation of the floating-point numbers inside both input AVX registers is applied to generate all necessary pairing of the input data for the update. The 16 update values are stored in 4 AVX registers and the update cycle is performed with each column of B, i.e., l times. Thus, intermediate results are always kept in registers and only the final values of U_i are written into memory.

To ensure efficient memory accesses we follow the rules listed below:

- The first element of B_1 is aligned on 128 byte boundary.
- During the partitioning of factors into supernodes, l is set to be a multiple of 4.
- l is set such that $4l$ double precision values fit "snugly" into the L1 cache memory and l^2 double precision values fit "snugly" into the L3 cache memory.

The above rules ensure that each memory read operation is aligned on 32 bytes and each $4 \times l$ block is aligned on the cache line. For highest efficiency, the process is implemented in assembly language. We allocated the 16 available AVX registers as follows:

- 4 registers accumulate the elements of U_i. These are updated by vector additions.
- 6 registers store intermediate results, computed by vector multiplications.
- The remaining 6 registers are used to preload and hold the input data.

A 3–way unrolling was necessary to circulate the registers through the process, and a further 4–way unrolling was implemented to keep the cycles on the 128 byte cache line boundary.

We can summarize the properties of our implementation as follows:

- 8 floating-point vector operations are performed for each pair of vector load operations.
- Loading data can be started an average of 10 processor cycles before needed.
- All load operations are aligned on 32 byte boundary.
- Cycles are aligned on cache line (128 byte) boundary, allowing prefetch instructions to hide cache latency.

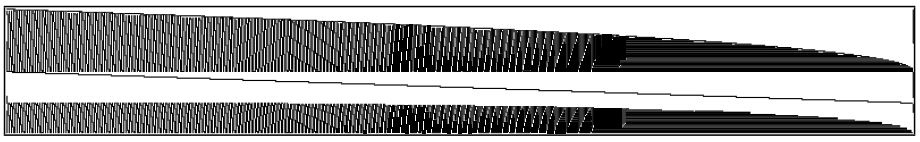
In our experiments we choose $l = 224$, which fulfilled all requirements noted above and proved to be efficient in our experiments. The most critical parts of our implementation were coded in assembly language, while the whole code was compiled and linked by the Intel C compiler version 11.1.072.

We would like to mention that BLAS [13] is a popular tool among the developers of mathematical software and it can achieve high performance. But, for us, it doesn't appear to be flexible enough to implement the data structures, described in this paper, to provide an efficient framework for using the 256 bit vector instructions.

4 Numerical results

In our numerical experiments we compared two generations of the Intel I7 processor family, the I7-2700K and I7-950 CPUs. The I7-2700K processor was run at 4.6 GHz, which is slightly above its 3.9 GHz "turbo" clockspeed. The times achieved on the I7-950 processor were scaled to compensate for its slightly lower clockspeed. Our machines were equipped with 16 gigabyte of main memory. In the experiments we used our IPM solver, called BPMPD [19].

FIGURE 1: Nonzero structure of problem NUG30



For detailed discussion we selected the well–known test problem nug30 [12]. We expect that on this problem the new vector instructions are very beneficial because the Cholesky factorization needed by our interior point implementation is very dense. The pattern of the problem matrix, and of $L + L^T$, are shown in Figures 1 and 2, and further details are summarized in Table 1.

FIGURE 2: Nonzero structure of $L + L^T$ on NUG30

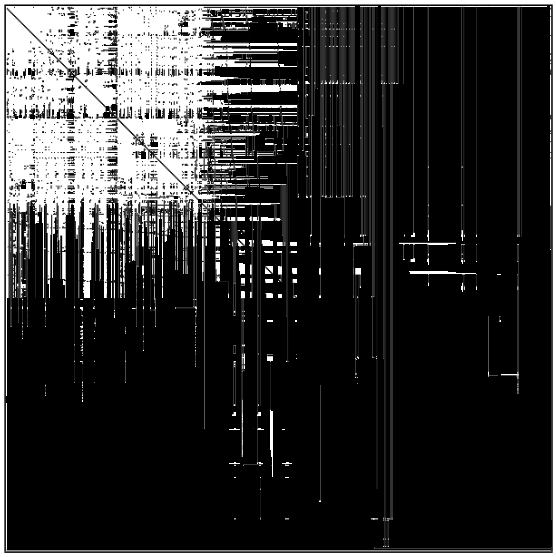


TABLE 1: Characteristics of problem NUG30

Number of rows in A	52260
Number of columns in A	379350
Nonzeroes in A	1567800
Nonzeroes in L	566532312
Flops to compute L	10296×10^9
Dimension of the dense window	24284
Flops in the dense window	4773×10^9
Flops in the sparse update	14×10^9

The efficiency of the different steps of the factorization is shown on Table 2. Figures given include the performance in the dense matrix–matrix multiplications in steps 3 and 4 of the algorithm, the efficiency in the whole dense Cholesky factorization part, the efficiency in the sparse update in step 5 and the performance of the whole factorization, including all necessary dense and sparse operations. Since both processors have 4 cores and hyperthreading capability, we executed our implementation in 1, 2, 4 and 8 threads. The experiments show that the matrix multiplications in steps 3 and 4 of our implemented algorithm benefit greatly from the vector and parallel processing, and the efficiency of the processing of the dense window is only marginally lower. The performance bottleneck of the process is the sparse update in step 5 where the speedup from multithreading is also lower, but since this step requires only a small portion of the total operations, it has a marginal effect on the overall factorization performance on this problem.

TABLE 2: Performance figures on NUG30 in gigaflops

Thre-	Multiplication		Dense window		Sparse update		Factorization	
ads	I7-950	I7-2700K	I7-950	I7-2700K	I7-950	I7-2700K	I7-950	I7-2700K
1	15.4	33.7	15.1	32.8	0.42	0.54	14.3	29.5
2	30.8	67.2	30.1	65.1	0.75	0.87	28.3	56.9
4	61.2	133.7	57.8	128.5	1.05	1.43	51.9	104.4
8	61.7	134.9	60.2	133.9	1.20	1.63	54.8	110.0

During the solution of this problem, 15 factorizations were computed. The performance of our implementation is compared on the two platforms in Table 3. We included the execution of the SSE2 code on both processors and the AVX version on the I7-2700K machine. All versions were run using 8 threads. Similar to the previous table, we scaled the I7-950 results to compensate for the lower clockspeed. The results show that the new generation CPU is about 20% faster than the older one and during the execution of the interior point code the AVX instruction set provides 70% speedup on this test problem.

TABLE 3: Total solution times and overall performance figures on NUG30

Platform	I7-950	I7-2700K/SSE2	I7-2700K/AVX
Time (sec).	2912 sec	2419 sec	1426 sec
Performance	53 Gflops	64 Gflops	108 Gflops
Relative	1	1.2	2.03

Finally, we present performance results of our implementation on other test cases. We have selected 9 large-scale linear programming problems from public sources and real-life applications [23, 12, 5, 4, 6]. Table 4 summarizes the characteristics of the test problems after our default preprocessing [22]. Figures given include the number of constraints, variables and nonzero elements in the test problems. The performance of our implementation on our I7-2700K machine is shown in Table 5, in terms of the number of nonzeroes in the Cholesky factorization (in thousands), the flops needed to compute one factorization (in billions), the solution time in seconds, and the sustained performance in gigaflops.

The results show that the implementation performs very well on problems that have a dense factorization, like *nug20* and *srd300*. On smaller problems (e.g. *dbic1*) or on larger problems with sparser factorization (e.g. *epa–10*) the overhead of the memory operations limits the performance of the floating-point unit.

5 Conclusions

We investigated how the AVX instruction set can be exploited in interior point methods. We developed techniques that help to exploit the new instruction set by decreasing data dependency and lowering the number of memory read operations. Our investigations showed that the improvements by the AVX instruction

TABLE 4: Problem statistics after presolve

Problem name	Rows in A	Colums in A	Nonzeroes in A
dbic1	33687	140358	781946
epa-10	333187	1321864	9525094
gasbaucp	128777	531061	1842988
l1_d10_40	74436	384798	1277562
nug20	15240	72600	304800
pds-100	97543	436003	1000207
rail4284	4176	1090526	11174639
spal-004	10203	321696	46167908
srd300	397580	1223770	58389690

TABLE 5: Performance results on large-scale test problems

Problem name	Nonzeroes in L ($\times 10^3$)	Flop count for one factorization ($\times 10^6$)	Solution time (in seconds)	Performance (in gigaflops)
dbic1	1896	185	3.9	2.0
epa-10	54608	16065	98.2	8.8
gasbaucp	24735	61770	106.2	52.5
l1_d10_40	69010	331572	120.5	60.8
nug20	46702	245169	47.8	85.9
pds-100	29325	41273	49.9	37.2
rail4284	5588	11115	24.6	15.6
spal-004	45269	264112	235.5	44.8
srd300	1143385	26062078	7889.0	109.2

set is mainly limited to the dense computational kernels and there are significant benefits only when necessary computations can be organized mostly into the dense kernels. On our test machine our implementation achieved 100 gigaflops sustained performance on some of the test problems, which shows the computational potential of the new architecture. We also concluded that the process speeds up well with multithreading. Although we had access to Intel processors only, we believe that the presented techniques work well on other platforms because the characteristics of modern multicore processors are similar.

5.1 Acknowledgements

This work was supported in part by Hungarian Research Fund OTKA K-111797. The author is indebted to the three anonymous referee for their helpful comments and remarks.

REFERENCES

1. I. Adler, N. Karmarkar, M.G.C. Resende, and G. Veiga. Data structures and programming techniques for the implementation of Karmarkar's algorithm. *ORSA J. on Computing*, 1(2):84–106, 1989.
2. R.P. Amestoy, T.A. Davis, and I.S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. on Matrix Analysis and Applications*, 17(4):886–905, 1996.
3. E.D. Andersen, J. Gondzio, C. Mészáros, and X. Xu. Implementation of interior point methods for large scale linear programs. In T. Terlaky, editor, *Interior Point Methods of Mathematical Programming*, pages 189–252. Kluwer Academic Publishers, 1996.
4. J.E. Beasley. Or-library is a collection of test data sets for a variety of or problems. World Wide Web, http://people.brunel.ac.uk/ ˜mastjjb/jeb/orlib/scpinfo.html.
5. W. Carolan, J. Hill, J. Kennington, S. Niemi, and S. Wichmann. An empirical evaluation of the korbx algorithms for military airlift applications. *Operations Research*, (38):240–248, 1990.
6. J. Castro. Recent advances in optimization techniques for statistical tabular data protection. *European J. on Operational Research*, (216):257–269, 2012.
7. A. George and J.W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Reviews*, 31:1–19, 1989.
8. J. Gondzio. Multiple centrality corrections in a primal-dual method for linear programming. *Computational Optimization and Applications*, 6:137–156, 1996.
9. J. Gondzio and T. Terlaky. A computational view of interior point methods for linear programming. In J. Beasley, editor, *Advances in Linear and Integer Programming*, pages 103–144. Oxford University Press, Oxford, England, 1995.
10. Intel Corporation. *Advanced Vector Extensions Programming Reference*, 2011.
11. Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference*, 2012.
12. S.E. Karisch and F. Rendl. Lower bounds for the quadratic assignment problem via triangle decompositions. *Mathematical Programming*, pages 137–152, 1995.
13. C.L. Lawson, R.J. Hanson, D. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979. Algorithm 539.
14. J.W.H. Liu, E.G. Ng, and B.W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. on Matrix Analysis and Applications*, 14(1):242–252, 1993.
15. I.J. Lustig, R.E. Marsten, and D.F. Shanno. The interaction of algorithms and architectures for interior point methods. In P.M. Pardalos, editor, *Advances in Optimization and Parallel Computing*, pages 190–205. Elsevier Sciences Publishers B.V., 1992.
16. I.J. Lustig, R.E. Marsten, and D.F. Shanno. Interior point methods for linear programming: Computational state of the art. *ORSA J. on Computing*, 6(1):1–15, 1994.
17. S. Mehrotra. High order methods and their performance. Technical Report 90-16R1, Department of Industrial Engineering and Managment Sciences, Northwestern University, Evanston, USA., 1991.
18. C. Mészáros. *The Efficient Implementation of Interior Point Methods for Linear Programming and their Applications*. PhD thesis, Eötvös Loránd University of Sciences, 1996.
19. C. Mészáros. The BPMPD interior-point solver for convex quadratic problems. *Optimization Methods and Software*, 11&12:431–449, 1999.
20. C. Mészáros. On the performance of the Cholesky factorization in interior point methods on Pentium 4 processors. *Central European Journal of Operations Research*, 13(4):289–298, 2005.
21. C. Mészáros. On the implementation of interior point methods for dual–core platforms. *Optimization Methods and Software*, 25(3):449–456, 2010.
22. C. Mészáros and U.H. Suhl. Advanced preprocessing techniques for linear and quadratic programming. *OR Spectrum*, 25:575–595, 2004.
23. H.D. Mittelmann and P. Spellucci. Decision tree for optimization software. World Wide Web, http://plato.la.asu.edu/guide.html, 2014.

24. D. Patterson and J. Hennessy. *Computer Organization and Design, 5th edition.* Elsevier Sciences Publishers B.V., 2013.
25. E. Rothberg and B. Hendrickson. Sparse matrix ordering methods for interior point linear programming. *INFORMS J. on Computing*, 10(1):107–113, 1998.
26. D.F. Shanno. Survey of implementation and computational experience with interior point methods. In *Interior Point Methods.* Eotvos Loránd University, Department of Operations Research, H-1088 Budapest, Múzeum krt. 6-8., Hungary, 1992.