Trends in Computational and Applied Mathematics, **24**, N. 2 (2023), 319-335
Sociedade Brasileira de Matemática Aplicada e Computacional
Online version ISSN 2676-0029
www.scielo.br/tcam
doi: 10.5540/tcam.2023.024.02.00319

The Impact of Compilation Flags and Choosing Single- or Double-Precision Variables in Linear Systems Solvers

R. C. BRUM[1], M. C. S. DE CASTRO[2] and C. O. FARIA[3*]

Received on December 10, 2018 / Accepted on September 28, 2022

ABSTRACT. This paper intends to show the impact of compiler optimization flags and the variable's precision on direct methods to solve linear systems. The chosen six methods are simple direct methods, so our work could be a study for new researchers in this field. The methods are LU Decomposition, LDU Decomposition, Gaussian Elimination, Gauss-Jordan Elimination, Cholesky Decomposition, and QR Decomposition using the Gram-Schmidt orthogonalization process. Our study showed a huge difference in time between single- and double-precision in all methods, but the error encountered in the single-precision was not so high. Also, the best flags to these methods were the '-O3' and the '-Ofast' ones.

Keywords: linear systems, compiler flags, optimization, variable precision.

1 INTRODUCTION

Many areas of science wants to find a fast and correct solution using computational models. At some point, most of the created models are turned into a system of linear equations. This system can be described as a set of m linear equations and n unknowns in which each equation can be described as $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$. All the coefficients can be stored in a matrix A and all constant terms in a column vector b. Both A and b can be stored in a single matrix M called by "augmented matrix", considering $m = n$, as described in (1.1).

$$M = \left[\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{array}\right] \tag{1.1}$$

*Corresponding author: Cristiane Oliveira Faria – E-mail: cofaria@ime.uerj.br
[1]Programa de Pós-Graduação em Ciências Computacionais, IME, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, RJ, Brazil – E-mail: rafinhacbrum@gmail.com https://orcid.org/0000-0003-3740-8379
[2]Programa de Pós-Graduação em Ciências Computacionais, IME, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, RJ, Brazil – E-mail: mariaclicia@gmail.com https://orcid.org/0000-0002-6315-1215
[3]Programa de Pós-Graduação em Ciências Computacionais, IME, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, RJ, Brazil – E-mail: cofaria@ime.uerj.br http://orcid.org/0000-0002-0402-7185

In this paper, we focus on six direct methods: Gaussian Elimination [14,17], Gauss-Jordan Elimination [17], LU Decomposition [10], LDU Decomposition [10], Cholesky Decomposition [17] and QR Decomposition using the Gram-Schmidt process [10]. All these methods are classified as direct methods and they find the solution in a finite number of steps [6]. More precisely, in Table 1 we present the quantity of floating-point operations (FLOPs) for each method, found in [8,11,15,16]

Table 1: Quantity of floating point operations needed by each method.

Method	# FLOPs	Reference
Gaussian Elimination	$O(n^3/3)$	(Trefethen, 1985) [16]
Gauss-Jordan Elimination	$O(n^3/2)$	(Meyer, 2000) [11]
LU Decomposition	$O(n^3/3)$	(Trahan *et al*, 2006) [15]
LDU Decomposition	$O(n^3/3)$	(Meyer, 2000) [11]
Cholesky Decomposition	$O(n^3/3)$	(Higham, 2009) [8]
QR Decomposition (using GS process)	$O(2n^3/3)$	(Trefethen, 1985) [16]

Although these methods were proposed to solve different matrices, this paper is focused on measuring the difference between single-precision variables and double-precision ones in a specific type of matrix to present a basic study to initial researchers. The variable precision defines how many digits can be stored for each floating-point number. Usually, a single-precision variable uses one word of 32 bits and a double-precision one uses two words of 32 bits each [12] which makes the computer takes more time to get it from memory.

Besides, some well-defined (rational) numbers in the decimal numerical system can be transformed into an irrational number in the binary numerical system, the one used by all computers. This mistranslated numbers together with the propagation errors of arithmetical operations can lead to huge numerical errors in the result. These are some of the motivations of the present work and we compare the quality of the results against the execution time to see which one is better. As our objective is to present a study to initial researches, we focused on executing the methods in common CPUs on contrary to recent research in FPGAs [4].

The outline of this work is as follows. Section 2 summarizes all options of compilers and optimization flags used here. Numerical experiments are reported in Section 3. Finally, some conclusions are given in Section 4.

2 PROGRAMMING LANGUAGE AND COMPILERS USED

In this paper, all methods were implemented in C Language, as it has static typing and we can manipulate the memory in run-time with dynamic allocation [3]. This language has two types of floating-point variables named `float` and `double` with single- and double-precision, respectively.

The C Language is a compiled language, and it means that our code has to be analyzed and transformed into binary code by a compiler [3]. In [5], they compare two compilers for C Language and one for C and C++ language. They concluded that the GCC, the Gnu compiler Collection (found in `https://gcc.gnu.org/`) were by far a better compiler than Microsoft Visual Studio. Therefore, two versions of GCC is used here, the 5.4.0 version (`https://gcc.gnu.org/onlinedocs/5.4.0/`) and the 7.1.0 version (`https://gcc.gnu.org/onlinedocs/7.1.0/`). The former version was released in June of 2016 and the latter one was released in May of 2017.

2.1 Compiler options used

The GCC compiler let us choose some options to optimize the compilation in many ways. The options are called compiler flags and there are eight ones to optimize: '-O0', '-O', '-O1', '-O2', '-O3', '-Ofast', '-Og' and '-Os'. Each one of these is at one level of optimization meaning that in each level all the optimization done in the previous level is done as well as new optimization. In [9], an algorithm to automatically choose the optimization option for programs. However, this is an iterative algorithm increasing the compilation time as well as they don't show the quality of the results. Now we will describe each one, for each version of GCC. All flags are explained in detail in [1, 2].

2.2 Compiler flags for GCC version 5.4.0

The default optimization flag is '-O0', which reduces compilation time and allows the debugging to show the expected results. The next level of optimization are the flags '-O' and '-O1', as they make the same optimization in the code. In this level, the compiler tries to minimize the code size and the execution time, but without time-consuming optimization options.

Continuing on the optimization levels, the '-O2' flag turns on almost all optimization options that do not have a space-speed trade-off in the compilation process. One example of optimization done in the '-O2' level is aligning (in memory) the beginning of a function to a power of 2 greater than a given number, usually a machine-dependent one. The next level of optimization is the '-03' flag. In this level, we have optimization turned to increase the code size to decrease the execution time. For example, the compiler copies the code of a function to where it finds this function callings, making inline functions.

The last level of optimization is the '-Ofast' flag. Besides enabling all optimization the '-O3' flag does, the '-Ofast' makes some optimization that is not valid for all standard-compliant programs. For example, it turns on some mathematical optimization that can lead to an incorrect result on programs that depend on some standard rules for math functions.

The last two flags, '-Os' and '-Og', is focused on optimizing the code size and code debugging, respectively. The '-Os' flag turns on all the '-O2' optimization that does not increase the final code size and some other specific optimization to decrease the size of the binary code. However, the '-Og' flag turns on only the optimization that does not interfere in the debugging process.

2.3 Compiler flags for GCC version 7.1.0

All the flags in the 7.1.0 version have the same optimization options as the 5.4.0 version with some new options. In the '-O' (or '-O1') level, one new optimization option is to reorder blocks of instructions to avoid branches and optimize code locality.

The '-O2' level has some new optimization as well as the old ones. In this version, the compiler search for stores in the memory that are smaller than a memory word and try to merge then into a single memory store. In the '-O3' level, one of the new optimization in this version is that the compiler tries to unroll loops that do not take many iterations to decrease the time consumed by controlling the program flow.

The other three flags, '-Ofast', '-Os' and '-Og', do not have any new optimization comparing to these flags in the 5.4.0 GCC version.

3 NUMERICAL TESTS AND RESULTS

In all experiments, the matrix A was created using $A = CC^T + nI$, having C as a square matrix of order n fulfill for random numbers between 0 and 999. We forced the linear system to have the trivial solution, that is, $\vec{b} = A(1,1,\ldots,1)^T$ being easier to compare the results obtained. This matrix A is well conditioned, as our objective is to compare the compilation flags in each method. Some explanation about error propagation and linear system sensitivity in [7]. The machine used in the experiments has an Intel Core i7 processor of 2.80 GHz, 4 cores and 8 GB of memory and 8 MB of cache memory, Ubuntu operational system version 16.04. [1]

3.1 Execution time considering all flags

Our first experiment was to execute all six methods with all flags to see the impact of each one. We tested with a linear system with 5000 unknowns. For the time experiment, we execute each combination of flag and method at least five times to get the average with a standard deviation of less than 1s. In Table 2, there is the average execution time for LU Decomposition in GCC version 5.4.0 and version 7.1.0, respectively. Comparing both versions, we can see that the most recent version of GCC does not have faster execution times than the version from 2016. One reason for this is because the GCC version 5.4.0 comes with the Ubuntu used in the tests and it can be optimized specifically for this version of OS. The GCC version 7.1.0 we had to install by ourselves. Another observation is that the fastest flag for LU Decomposition was the flag '-O3'. When the '-Ofast' flag was used, the execution time in the newest version increased compared to '-03' flag, which was a little unexpected as in the former flag the compiler makes the program skips some comparison in the maths.

Table 3 shows the average execution time of LDU Decomposition in versions of GCC version 5.4.0 and version 7.1.0. The observations we can make from this table is that the fastest flag for LDU Decomposition in both GCC versions is the '-O3'. We can also observe that the '-Os'

[1] Our source code is available at `https://github.com/rafaelaBrum/direct-methods`

Table 2: Execution times of LU Decomposition - matrix with 5000 unknowns.

	GCC version 5.4.0		GCC version 7.1.0	
Flag	Single-precision	Double-precision	Single-precision	Double-precision
-O0	284.24 s	290.12 s	286.43 s	296.84 s
-O	108.49 s	118.74 s	108.24 s	117.98 s
-O1	108.43 s	118.80 s	107.69 s	118.04 s
-O2	59.12 s	80.34 s	59.10 s	76.65 s
-O3	46.29 s	78.98 s	45.91 s	70.99 s
-Ofast	46.85 s	78.96 s	61.40 s	79.57 s
-Og	106.61 s	118.12 s	107.63 s	117.78 s
-Os	77.27 s	79.53 s	64.14 s	79.33 s

Table 3: Execution times of LDU Decomposition - matrix with 5000 unknowns.

	GCC version 5.4.0		GCC version 7.1.0	
Flag	Single-precision	Double-precision	Single-precision	Double-precision
-O0	284.45 s	290.67 s	288.24 s	297.53 s
-O	107.82 s	113.24 s	108.07 s	119.13 s
-O1	107.86 s	121.35 s	108.10 s	119.16 s
-O2	59.31 s	71.79 s	59.47 s	80.94 s
-O3	46.17 s	64.99 s	46.55 s	80.10 s
-Ofast	47.45 s	79.48 s	52.65 s	114.12 s
-Og	109.71 s	117.34 s	107.40 s	118.95 s
-Os	74.91 s	86.43 s	63.69 s	87.22 s

and '-Ofast' flag have the most meaningful time variations between versions due to the different optimizations made in each version of the compiler.

Comparing the times between LU and LDU Decomposition, we can see that they are very similar, as the algorithms are almost the same, only having one more step in the final decomposition.

Table 4 shows the average execution time of Gaussian Elimination in both versions of GCC, version 5.4.0 and version 7.1.0. For the Gaussian Elimination, the fastest flag is not the '-O3' or the '-Ofast', but it is the '-0g' one, the optimization level focused on debugging having the same time as the first level of optimization. As the '-03' flag turns on the optimization that increases the code, for example using inline functions, it may increase the number of variables used as well as the memory access. Finally, the '-Os' flag, the optimization level focused on not increase the final code size, has a bad execution time in the Gaussian Elimination. The main part of this algorithm is a big loop. As to optimize a loop we have to increase the size of the code, it is expected that the '-Os' optimization level takes more time to finish that other levels of optimization.

Table 4: Execution times of Gaussian Elimination - matrix with 5000 unknowns.

	GCC version 5.4.0		GCC version 7.1.0	
Flag	Single-precision	Double-precision	Single-precision	Double-precision
-O0	410.45 s	413.15 s	417.31 s	421.74 s
-O	139.99 s	147.88 s	139.67 s	149.60 s
-O1	139.30 s	147.93 s	139.74 s	150.08 s
-O2	205.27 s	316.08 s	205.34 s	317.09 s
-O3	204.88 s	316.82 s	205.79 s	317.40 s
-Ofast	205.56 s	316.90 s	207.51 s	317.12 s
-Og	139.93 s	146.45 s	138.20 s	147.11 s
-Os	204.24 s	316.86 s	204.45 s	316.87 s

Table 5: Execution times of Gauss-Jordan Elimination - matrix with 5000 unknowns.

	GCC version 5.4.0		GCC version 7.1.0	
Flag	Single-precision	Double-precision	Single-precision	Double-precision
-O0	799.43 s	823.06 s	821.80 s	833.27 s
-O	342.39 s	467.30 s	338.41 s	463.62 s
-O1	342.83 s	466.87 s	340.40 s	463.72 s
-O2	323.91 s	441.57 s	323.96 s	441.29 s
-O3	323.88 s	441.49 s	324.43 s	442.05 s
-Ofast	322.22 s	442.22 s	322.51 s	442.19 s
-Og	342.02 s	463.40 s	340.54 s	464.37 s
-Os	323.45 s	441.58 s	317.42 s	440.68 s

In Table 5, there is the average execution time for Gauss-Jordan Elimination in GCC version 5.4.0 and version 7.1.0, respectively. These results show that this algorithm cannot be optimized as the Gaussian Elimination, for example. We observe a decrease in the execution time in Gauss-Jordan Elimination with the first level of optimization, the '-O' flag, compared to the default optimization level ('-O0' flag), but we don't see any decrease in the other levels as the Gauss-Jordan algorithm is too sequential and dependent.

Comparing Gaussian Elimination to Gauss-Jordan Elimination, we can see that the Gaussian Elimination is a better option as it consumes less time to execute and can be optimized a little more. From the no optimization level ('-O0' flag) to the first level of optimization ('-O'), the Gaussian Elimination reduces its execution time by a factor of 3 and the Gauss-Jordan Elimination reduces its time by a factor of 2.35.

Table 6 shows the average execution time of Cholesky Decomposition in versions of GCC version 5.4.0 and 7.1.0. This method is the one that has the second most gain in the optimization levels. For the first level, the execution time is reduced to a third of the original time. In the '-Ofast'

time, the fastest optimization level to Cholesky Decomposition, it is reduced to a sixth of the original time.

Table 6: Execution times of Cholesky Decomposition - matrix with 5000 unknowns.

	GCC version 5.4.0		GCC version 7.1.0	
Flag	Single-precision	Double-precision	Single-precision	Double-precision
-O0	122.38 s	124.96 s	121.85 s	124.12 s
-O	30.60 s	33.93 s	30.17 s	33.66 s
-O1	30.51 s	33.91 s	30.17 s	33.62 s
-O2	30.47 s	34.04 s	30.14 s	33.81 s
-O3	30.54 s	34.30 s	30.22 s	33.35 s
-Ofast	19.55 s	40.95 s	19.68 s	34.81 s
-Og	47.80 s	49.64 s	47.55 s	49.64 s
-Os	31.34 s	35.05 s	30.98 s	35.70 s

Table 7: Execution times of QR Decomposition - matrix with 5000 unknowns.

	GCC version 5.4.0		GCC version 7.1.0	
Flag	Single-precision	Double-precision	Single-precision	Double-precision
-O0	970.45 s	978.21 s	972.52 s	985.82 s
-O	202.20 s	215.15 s	203.25 s	219.55 s
-O1	202.25 s	215.15 s	203.21 s	219.49 s
-O2	202.23 s	213.81 s	234.20 s	238.09 s
-O3	168.37 s	228.64 s	188.12 s	212.06 s
-Ofast	81.72 s	149.63 s	78.63 s	152.67 s
-Og	224.93 s	238.42 s	224.26 s	245.42 s
-Os	224.47 s	234.26 s	223.36 s	240.93 s

Table 7 shows the average execution time of QR Decomposition using the Gram-Schmidt process in both versions of GCC, version 5.4.0 and version 7.1.0. The QR Decomposition method is the one that takes more time, as it is the most complex algorithm. It has to iterate the matrix three times: one to create the orthogonal matrix Q; the second to multiply the Q^T to $\vec{b}$; and the last one to find the value of the unknowns. As it used many Maths operations all over the algorithm [10], this method is the one that gains most in the optimization process. With the '-Ofast' flag, it reduces the execution time by a factor of 12.

In summary, with all experiments in this section, we conclude that only with the first level of optimization, the '-O' flag, the execution time of all methods drops to at least half of the execution time with no optimization (flag '-O0'). Also, we can regard similar execution times from flags '-O' and '-O1' as they represent the same optimization level.

3.2 Numerical solution considering all flags

The next test was concerning the numerical results obtained by each flag. To this test, we used the same linear system from the previous test but we took the results and calculated the absolute error of the obtained results.

Although the average execution time differed between both versions of GCC, the numerical results were the same. Hence, we will show the euclidean norm of errors for just the 5.4.0 GCC version.

Table 8 shows the euclidean norm of errors for the results obtained through the LU Decomposition. We can see in this table that no flag changed the precision of the results. So, for the LU Decomposition, we can choose the best optimization level looking only at the execution time in Table 2. Having said that, the best flag for LU Decomposition is the '-Ofast' for GCC version 5.4.0 and '-O3' for GCC version 7.1.0.

Besides, the double-precision variables have an absolute error of 10^{-13} and the single-precision has one of 10^{-4}. The latter precision has an absolute error that can be not acceptable in some applications.

Table 8: Absolute error of results from LU Decomposition - matrix with 5000 unknowns.

Flag	Single-precision	Double-precision
-O0	1.50×10^{-4}	1.67×10^{-13}
-O	1.50×10^{-4}	1.67×10^{-13}
-O1	1.50×10^{-4}	1.67×10^{-13}
-O2	1.50×10^{-4}	1.67×10^{-13}
-O3	1.50×10^{-4}	1.67×10^{-13}
-Ofast	1.50×10^{-4}	1.67×10^{-13}
-Og	1.50×10^{-4}	1.67×10^{-13}
-Os	1.50×10^{-4}	1.67×10^{-13}

For LDU Decomposition the results in both precisions were the same as in LU Decomposition. Then, the decision for the best flag is made again looking for the execution time, in Table 3. For the 2016 version of GCC, the best flags are '-O3' and '-Ofast', as they have almost the same execution time. For the other version of GCC, the best flag is '-O3'. As the absolute error is the same as LDU Decomposition, the single-precision one can again be not acceptable in some applications as it is 10^{-4}.

The next table (Table 9) shows the euclidean norm of errors of the results obtained through the Gaussian elimination.

As we can see in this table, some flags could not calculate the final result due to the number of operations to find the unknowns. It probably caused an overflow (or underflow) which prints '-NaN' as a result. Having said that, the only four flags that we can choose to be the best ones to

Table 9: Absolute error of results from Gaussian Elimination - matrix with 5000 unknowns.

Flag	Single-precision	Double-precision
-O0	-NaN	-NaN
-O	-NaN	-NaN
-O1	-NaN	-NaN
-O2	1.50×10^{-4}	1.67×10^{-13}
-O3	1.50×10^{-4}	1.67×10^{-13}
-Ofast	1.50×10^{-4}	1.67×10^{-13}
-Og	-NaN	-NaN
-Os	1.50×10^{-4}	1.67×10^{-13}

the Gaussian elimination is '-O2', '-O3', '-Ofast' and '-Os' flags. Looking at the execution times in Table 4, we conclude that all these four flags can be the best as they all have similar execution times.

Besides, in concern of the precision, the single-precision has, again, an absolute error of 10^{-4}, which can be bad for some applications. So, the double-precision variables, which has an error of 10^{-13}, would be the best choice.

Table 10 shows the euclidean norm of errors of the Gauss-Jordan elimination and we can see a similar behaviour as the Gaussian elimination (Table 9). As the Gauss-Jordan elimination transforms the matrix M (shown in 1.1) into a canonical matrix and the step of finding the unknowns does not have any operation of division or multiply. It explains why in the Gauss-Jordan elimination we can get a result for all flags while in Gaussian elimination we cannot get with the double-precision variables. However, the best flags concerning precision are the same as Gaussian elimination: '-O2', '-O3', '-Ofast' and '-Os'. When looking at execution times (Table 5), we conclude that all four flags can be used to optimize this method as they have similar execution times.

Table 10: Absolute error of results from Gauss-Jordan Elimination - matrix with 5000 unknowns.

Flag	Single-precision	Double-precision
-O0	-NaN	1.78×10^{-2}
-O	-NaN	1.78×10^{-2}
-O1	-NaN	1.78×10^{-2}
-O2	1.50×10^{-4}	1.67×10^{-13}
-O3	1.50×10^{-4}	1.67×10^{-13}
-Ofast	1.50×10^{-4}	1.67×10^{-13}
-Og	-NaN	1.78×10^{-2}
-Os	1.50×10^{-4}	1.67×10^{-13}

The single-precision in this method in the four flags mentioned above has the euclidean norm of errors of 10^{-4}, which can be not acceptable in some applications. Then, the best option for Gauss-Jordan elimination is double-precision. We can observe that in the other four flags, the norms of the double-precision variables are big and similar, because the input data has big numbers and in the operations of multiply and divide the computer has to truncate or round somewhat often.

For the Cholesky Decomposition, we observed that all flags calculated the results with the same error Euclidean norm of 1.50×10^{-4} for single-precision and 3.55×10^{-15} for double-precision. So, we can choose the best flags to use looking only at the execution times (Table 6). The best flag for Cholesky Decomposition is the '-Ofast' one to single-precision variables and the '-O3' one to double-precision.

As for the QR Decomposition, the Euclidean error norms observed was 2.12×10^{-4} to the single-precision and 1.18×10^{-13} to the double-precision to almost all flags except the '-Ofast' one (equal 1.47×10^{-4} to the single-precision and 1.02×10^{-13} to the double-precision). This method is the one with the biggest error norm due to the number of multiplications and divisions made, which leads to many truncate and rounding errors, as said in [13].

Most of all flags in the QR Decomposition don't change the norm of errors except the '-Ofast' one. With this in mind and looking at the execution times in Table 7, we conclude that the '-Ofast' flag is the best optimization choice for the QR Decomposition method.

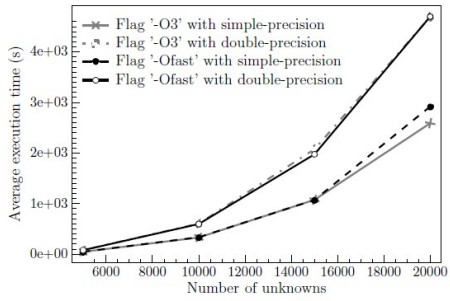
To finalize these tests, we can see that most of the methods have the '-O3' or the '-Ofast' flag being the best option for optimization. Having said that, we show our final test, with bigger matrices.

3.3 Comparison between double- and single-precision

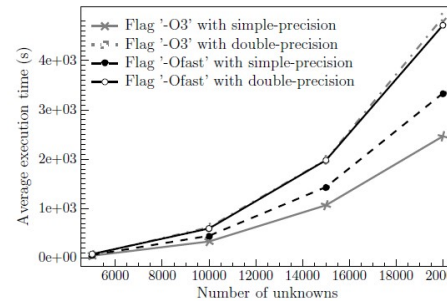
Our next test was executed with the '-O3' and '-Ofast' flags and bigger matrices, with 5000, 10000, 15000 and 20000 unknowns. All the systems were created with the same methodology shown at the beginning of Section 3. These tests were made in both versions of GCC, the 5.4.0 and the 7.1.0. Each combination of flag and method was executed at least 10 times to get a standard deviation of less than 5%.

Figure 1 shows the average execution time for LU Decomposition for GCC version 5.4.0 and version 7.1.0, respectively. As the input data increases in size, the absolute difference between single- and double-precision increases, but it is not linear. Table 11 shows how much the double-precision execution is slower than the single-precision. We computed this relation using E_d/E_s, where E_d is the execution time for double-precision variables and E_s is the one for single-precision variables.

For the '-O3' flag, for example, the Table 11 means that with 5000 unknowns the double-precision time is 1.706 times bigger in the GCC version 5.4.0 and 1.546 times bigger in the 7.1.0 version than the single-precision time; with 10000 unknowns the double-precision execution time is 1.832 times bigger in the former version and 1.864 times in the other version than



(a) GCC version 5.4.0



(b) GCC version 7.1.0

Figure 1: Average times of LU Decomposition.

Table 11: Relation between double- and single-precision times of LU Decomposition.

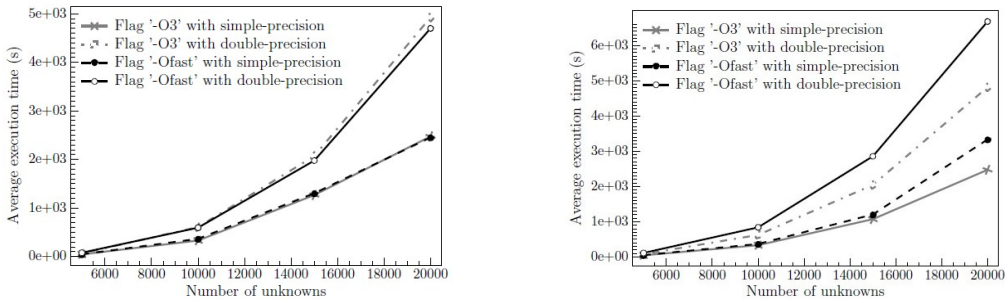
	Flag '-O3'		Flag '-Ofast'	
# of unknowns	GCC v. 5.4.0	GCC v. 7.1.0	GCC v. 5.4.0	GCC v. 7.1.0
5000	1.706	1.546	1.685	1.296
10000	1.832	1.864	1.780	1.330
15000	1.930	1.862	1.841	1.378
20000	1.813	1.977	1.609	1.413

the single-precision time. For 15000 unknowns, the double-precision time is 1.930 times bigger in the oldest version and 1.862 times bigger in the newest version than the single-precision time; for 20000 unknowns, the double-precision time is 1.813 times bigger in GCC version 5.4.0 and 1.977 bigger in GCC version 7.1.0 than the single-precision.

Figure 2 shows the average execution time for LDU Decomposition for GCC version 5.4.0 and version 7.1.0, respectively. As the input data increases in size, the absolute difference between single- and double-precision increases, but it is again not linearly. In the next table (Table 12) we can see how much the double-precision execution is slower than the single-precision.

Table 12: Relation between double- and single-precision times of LDU Decomposition.

	Flag '-O3'		Flag '-Ofast'	
# of unknowns	GCC v. 5.4.0	GCC v. 7.1.0	GCC v. 5.4.0	GCC v. 7.1.0
5000	1.408	1.721	1.675	2.168
10000	1.835	1.870	1.628	2.271
15000	1.633	1.905	1.517	2.370
20000	1.990	1.947	1.914	2.003

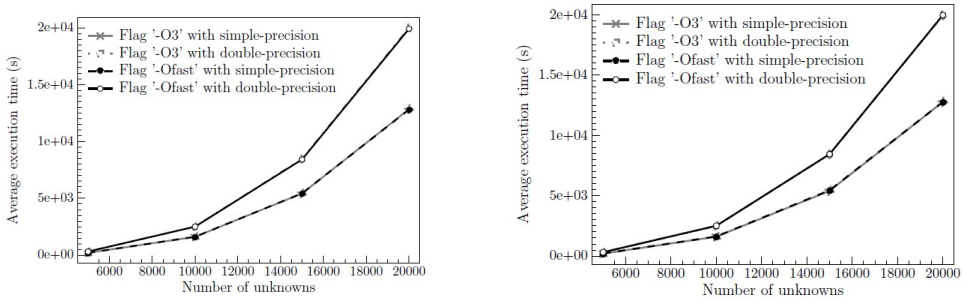


(a) GCC version 5.4.0 (b) GCC version 7.1.0

Figure 2: Average times of LDU Decomposition.

For the '-Ofast' flag, for example, the Table 12 means that with 5000 unknowns the double-precision time is 1.675 times bigger in the GCC version 5.4.0 and 2.168 times bigger in the 7.1.0 version than the single-precision time; with 10000 unknowns the double-precision execution time is 1.628 times bigger in the former version and 2.271 times in the other version than the single-precision time. For 15000 unknowns, the double-precision time is 1.517 times bigger in the oldest version and 2.370 times bigger in the newest version than the single-precision time; for 20000 unknowns, the double-precision time is 1.914 times bigger in GCC version 5.4.0 and 2.003 times bigger in GCC version 7.1.0 than the single-precision. In the GCC version 7.1.0, the double-precision code of LDU Decomposition takes more than twice longer to finish than the single-precision one.

The graphics in Figure 3 show the average execution time for Gaussian Elimination for GCC version 5.4.0 and version 7.1.0. As the input data increases in size, the absolute difference between single- and double-precision increases linearly. In the next table (Table 13) we can see how much the double-precision execution is slower than the single-precision.



(a) GCC version 5.4.0 (b) GCC version 7.1.0

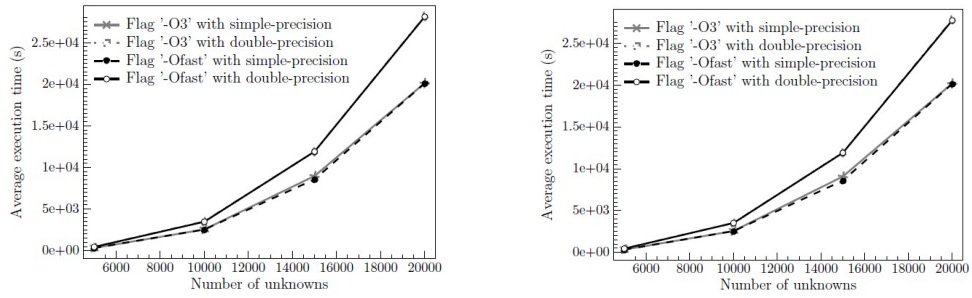
Figure 3: Average times of Gaussian Elimination.

Table 13: Relation between double- and single-precision times of Gaussian Elimination.

	Flag '-O3'		Flag '-Ofast'	
# of unknowns	GCC v. 5.4.0	GCC v. 7.1.0	GCC v. 5.4.0	GCC v. 7.1.0
5000	1.546	1.542	1.542	1.528
10000	1.557	1.559	1.554	1.546
15000	1.561	1.561	1.547	1.549
20000	1.558	1.562	1.555	1.563

For the '-O3' flag, for example, the Table 13 means that with 5000 unknowns the double-precision time is 1.546 times bigger in the GCC version 5.4.0 and 1.542 times bigger in the 7.1.0 version than the single-precision time; with 10000 unknowns the double-precision execution time is 1.557 times bigger in the former version and 1.559 times in the other version than the single-precision time. For 15000 unknowns, the double-precision time is 1.561 times bigger in the oldest version and 1.561 times bigger in the newest version than the single-precision time; for 20000 unknowns, the double-precision time is 1.558 times bigger in GCC version 5.4.0 and 1.562 times bigger in GCC version 7.1.0 than the single-precision. As we can see in Table 13, all double-precision times are around 1.5 times longer than the single-precision times.

The graphics in Figure 4 show the average execution time for Gauss-Jordan Elimination for GCC version 5.4.0 and version 7.1.0, respectively. As the input data increases in size, the absolute difference between single- and double-precision increases linearly again. In the next table (Table 14) we can see how much the double-precision execution is slower than the single-precision.



(a) GCC version 5.4.0

(b) GCC version 7.1.0

Figure 4: Average times of Gauss-Jordan Elimination.

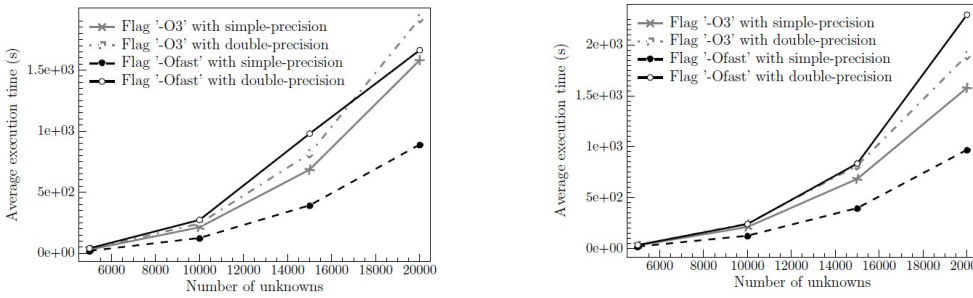
For the '-Ofast' flag, for example, the Table 14 means that with 5000 unknowns the double-precision time is 1.372 times bigger in the GCC version 5.4.0 and 1.371 times bigger in the 7.1.0 version than the single-precision time; with 10000 unknowns the double-precision execution time is 1.369 times bigger in the former version and 1.382 times in the latter version than the single-precision time. For 15000 unknowns, the double-precision time is 1.390 times bigger in the oldest version and 1.389 times bigger in the newest version than the single-precision time;

Table 14: Relation between double- and single-precision times of Gauss-Jordan Elimination.

	Flag '-O3'		Flag '-Ofast'	
# of unknowns	GCC v. 5.4.0	GCC v. 7.1.0	GCC v. 5.4.0	GCC v. 7.1.0
5000	1.363	1.363	1.372	1.371
10000	1.379	1.376	1.369	1.382
15000	1.328	1.313	1.390	1.389
20000	1.398	1.377	1.398	1.376

for 20000 unknowns, the double-precision time is 1.398 times bigger in GCC version 5.4.0 and 1.376 times bigger in GCC version 7.1.0 than the single-precision.

Figure 5 shows the average execution time for Cholesky Decomposition for GCC version 5.4.0 and version 7.1.0, respectively. As the input data increases in size, the absolute difference between single- and double-precision increases, but it is not linearly. In the next table (Table 15) we can see how much the double-precision execution is slower than the single-precision.



(a) GCC version 5.4.0

(b) GCC version 7.1.0

Figure 5: Average times of Cholesky Decomposition.

Table 15: Relation between double- and single-precision times of Cholesky Decomposition.

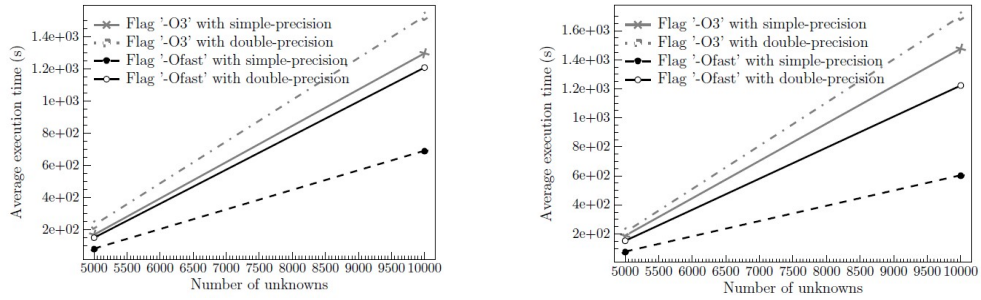
	Flag '-O3'		Flag '-Ofast'	
# of unknowns	GCC v. 5.4.0	GCC v. 7.1.0	GCC v. 5.4.0	GCC v. 7.1.0
5000	1.123	1.104	2.095	1.769
10000	1.143	1.156	2.177	1.903
15000	1.191	1.192	2.495	2.104
20000	1.215	1.201	1.868	2.368

For the '-O3' flag, for example, the Table 15 means that with 5000 unknowns the double-precision time is only 1.123 times bigger in the GCC version 5.4.0 and 1.104 times bigger in the 7.1.0 version than the single-precision time; with 10000 unknowns the double-precision execution time is 1.143 times bigger in the former version and 1.156 times in the latter version than

the single-precision time. For 15000 unknowns, the double-precision time is 1.191 times bigger in the oldest version and 1.192 times bigger in the newest version than the single-precision time; for 20000 unknowns, the double-precision time is 1.215 times bigger in GCC version 5.4.0 and 1.201 times bigger in GCC version 7.1.0 than the single-precision.

In the Cholesky Decomposition, there is a huge difference in the relation shown in Table 15 between the '-O3' and '-Ofast' times. In the '-O3' optimization flag, both precisions have similar times. In contrast, with the '-Ofast' flag the double-precision times is around twice the single-precision times.

Figure 6 shows the average execution time for QR Decomposition for GCC version 5.4.0 and version 7.1.0, respectively. The QR Decomposition has the biggest complexity of all methods, $O(N^3)$, and that why the time to 10000 unknowns is eight times bigger than the time to 5000 unknowns. For that reason, the other two input sizes were not computed. In the next table (Table 16) we can see how much the double-precision execution is slower than the single-precision.



(a) GCC version 5.4.0

(b) GCC version 7.1.0

Figure 6: Average times of QR Decomposition.

Table 16: Relation between double- and single-precision times of QR Decomposition.

	Flag '-O3'		Flag '-Ofast'	
# of unknowns	GCC v. 5.4.0	GCC v. 7.1.0	GCC v. 5.4.0	GCC v. 7.1.0
5000	1.358	1.127	1.831	1.942
10000	1.177	1.152	1.748	2.022

For the '-Ofast' flag, for example, the Table 16 means that with 5000 unknowns the double-precision time is 1.831 times bigger in the GCC version 5.4.0 and 1.942 times bigger in the 7.1.0 version than the single-precision time; with 10000 unknowns the double-precision execution time is 1.748 times bigger in the former version and 2.022 times in the latter version than the single-precision time.

Similar to the Cholesky Decomposition, the QR Decomposition has big differences between the relation in the two flags studied. For the '-03' flag, both precisions have similar execution

times. However, for the '-Ofast' flag, the double-precision execution time is almost twice the single-precision execution time.

4 CONCLUSIONS

Our computational experiments show some difference in the execution time of some methods with double-precision and single-precision variables. Besides, we search for the best optimization flag in GCC for each of six different methods. The methods studied was: LU Decomposition, LDU Decomposition, Gaussian Elimination, Gauss-Jordan Elimination, Cholesky Decomposition and QR Decomposition using the Gram-Schmidt process. We showed execution times for all these methods in six optimization flags of GCC compiler: '-O0', '-O', '-O1', '-O2', '-O3', '-Ofast', '-Og', '-Os'. Besides, the experiments were compiled with two versions of GCC, version 5.4.0, released in June of 2016, and version 7.1.0, released in May of 2017.

Our first conclusion is that the best optimization flags for the studied methods were '-O3' or '-Ofast', depending on the method. The '-O3' turns on all optimization from two previous levels and optimization related to spend more memory or time compiling to decrease execution time. The '-Ofast' flag turn on all optimization from '-O3' level (the ones mentioned before) and some optimization related to the mathematical functions.

Finally, between single- or double-precision variables, we observed that in some methods (like Gauss-Jordan elimination) the double-precision time can be more than 1.5 times the single-precision one and the single-precision error is small (the biggest error is around 10^{-3}) for the flags mentioned above. If the application in question is not real-time or it has tolerance of 10^{-3} or bigger, it is better to use single-precision variables. However, if the application has to have a smaller tolerance, it is better to use a double-precision variable.

Acknowledgments

This work was partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and by Fundação de Amparo a Pesquisa do Estado do Rio de Janeiro (FAPERJ), process number E-26/010.001222/2019 and E-26/200.934/2017

REFERENCES

[1] Using the GNU Compiler Collection (GCC) (2016). URL `https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/`. Accessed December 6, 2018.

[2] Using the GNU Compiler Collection (GCC) (2017). URL `https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/`. Accessed December 6, 2018.

[3] A.F.G. Ascencio & E.A.V.d. Campos. "Fundamentos da programação de computadores: algoritmos, Pascal, C/C++ e Java". Pearson Prentice Hall, São Paulo (2008).

[4] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek & S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, **180**(12) (2009), 2526–2533. doi:10.1016/j.cpc.2008.11.005. URL `http://dx.doi.org/10.1016/j.cpc.2008.11.005`.

[5] T. Botor & H. Habiballa. Compiler optimization for scientific computation in C/C++. In "Proceedings of the International Conference of Computational Methods in Sciences and Engineering 2018 (ICCMSE 2018)". Thessaloniki, Greece (2018), p. 030004. doi:10.1063/1.5079067. URL `http://aip.scitation.org/doi/abs/10.1063/1.5079067`.

[6] M.C.C. Cunha. "Métodos numéricos". Editora da UNICAMP (2003).

[7] A. Deif. "Sensitivity analysis in linear systems". Springer Science & Business Media (2012).

[8] N.J. Higham. Cholesky factorization. *Wiley Interdisciplinary Reviews: Computational Statistics*, **1**(2) (2009), 251–254. doi:10.1002/wics.18.

[9] K. Hoste & L. Eeckhout. Cole: compiler optimization level exploration. In "Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization". ACM (2008), p. 165–174.

[10] S. Lipschutz. "Álgebra linear (4a. ed.).". Grupo A - Bookman (2000). URL `http://public.eblib.com/choice/publicfullrecord.aspx?p=3236279`. OCLC: 923757758.

[11] C.D. Meyer. "Matrix analysis and applied linear algebra", volume 71. Siam (2000).

[12] D.A. Patterson & J.L. Hennessy. "Organização e projeto de computadores: interface hardware/software". Elsevier Brasil (2014).

[13] M.A.G. Ruggiero & V.L.d.R. Lopes. "Cálculo numérico: aspectos teóricos e computacionais". Makron Books do Brasil (1997).

[14] R. Sedgewick. "Algorithms in C". Addison-Wesley series in computer science. Addison-Wesley Pub. Co, Reading, Mass (1990).

[15] J. Trahan, A. Kaw & K. Martin. Computational time for finding the inverse of a matrix: LU decomposition vs. naive gaussian elimination. *University of South Florida*, (2006).

[16] L.N. Trefethen. Three mysteries of Gaussian elimination. *ACM SIGNUM Newsletter*, **20**(4) (1985), 2–5. doi:10.1145/1057954.1057955.

[17] D.S. Watkins. "Fundamentals of matrix computations". Pure and applied mathematics. Wiley-Interscience, New York, 2nd ed ed. (2002).