Cholesky QR Factorization computational performance study on CPU and GPU

Numerical Algorithms course
Department of Environmental Sciences, Informatics and Statistics
Ca' Foscari University of Venice

Alessandro Bicciato*

Contents

1 Introduction **2**

2 Mathematical tools **3**
2.1 Cholesky-QR factorization 3
2.2 Cholesky decomposition 3
2.3 QR factorization 3

3 Algorithms **4**
3.1 Cholesky-QR 4
3.2 Classical Gram-Schmidt 4
3.3 Modified Gram-Schmidt 4
3.4 Singular value QR 5
3.5 Communication-avoiding QR 5

4 Results of the study **6**
4.1 Intel i7 - NVidia GeForce 640M LE test 6
4.2 Intel i5 - NVidia GeForce 1070 test 8

5 Conclusions **10**

*Student id: 845487, E-Mail: 845487@stud.unive.it

1 Introduction

In this paper we will use MatLAB to do a performance comparison of different matrices factorization methods, such as native QR function, the hybrid Cholesky QR method (CholQR), Gram-Schmidt classical method (CGS), Gram-Schmidt modified method (MGS), singular value QR method (SVQR) and Communication-avoiding QR method (CAQR).

Alongside the comparison between these methods, we will compare also the computational difference between using data structures for the CPU and for the GPU. The data structures used are all native of MatLAB core (for CPU) and Parallel Computing Toolbox add-on (for GPU).

We programmed two tests in MatLAB, version R2018a. For the first trial we used a laptop with processor Intel i7-3520M turbo boosted at 3.40 GHz, 8 GB of RAM and graphic card NVidia GeForce GT 640M LE 1 GB. For the second trial we used a desktop computer with processor Intel i5-7600 at 3.50 GHz, 16 GB of RAM and graphic card NVidia GeForce GTX 1070 6 GB. As MatLAB uses NVidia CUDA libraries to compute on the GPU, only NVidia graphic cards are compatible with the software; the CUDA compute capability declared of the hardware used is 3.0 for laptop card and 6.1 for the desktop one.

This paper is meant to be a study in a broad sense based on the results of the article published in 2015 *Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs* from Ichitaro Yamazaki[1], Stanimire Tomov[1] and Jack Dongarra[1].

As the tools were limited and the authors haven't yet released more specific algorithms, the study will focus, as said earlier, only on the part of matrix factorization and only with high level of programming using a single GPU.

The paper is organized in three sections. Section 2 will briefly introduce the mathematical tools used, Section 3 will have the pseudo-code of the various methods and Section 4 will contain the results of the tests.

[1]Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996 (iyamazak@eecs.utk.edu, tomov@eecs.utk.edu, dongarra@eecs.utk.edu).

2 Mathematical tools

2.1 Cholesky-QR factorization

The Cholesky QR method is the result of the derivation of a block QR factorization based on the Cholesky Factorization.

If $S = W^T W = R^T R$, where R is the Cholesky factor, $Q = WR^{-1}$ defines the QR factorization for W. Although this method is rarely or never used computationally, it has some attractive characteristics; it is a QR factorization, it is based on a level 3 BLAS kernel and a triangular system solution, it involves only 50% more arithmetic than Gram-Schmidt, and it also requires one synchronization point in parallel implementations. Researchers have noticed that it is not as stable as modified Gram-Schmidt, but frequently it can be more stable than Gram-Schmidt. One of the problematic issues with this method is that the more ill-conditioning S is, the less stable the Cholesky factorization becomes.

2.2 Cholesky decomposition

The Cholesky decomposition of a Hermitian positive-definite matrix A is a decomposition of the form

$$A = LL^*$$

where L is a lower triangular matrix with real and positive diagonal entries, and L^* denotes the conjugate transpose of L. Every Hermitian positive-definite matrix (and thus also every real-valued symmetric positive-definite matrix) has a unique Cholesky decomposition. If the matrix A is Hermitian and positive semi-definite, then it still has a decomposition of the form $A = LL^*$ if the diagonal entries of L are allowed to be zero. When A has only real entries, L has only real entries as well, and the factorization may be written $A = LL^T$.

The Cholesky decomposition is unique when A is positive definite; there is only one lower triangular matrix L with strictly positive diagonal entries such that $A = LL^*$. However, the decomposition need not be unique when A is positive semidefinite. The converse holds trivially: if A can be written as LL^* for some invertible L, lower triangular or otherwise, then A is Hermitian and positive definite.

2.3 QR factorization

Any real square matrix A may be decomposed as

$$A = QR,$$

where Q is an orthogonal matrix (its columns are orthogonal unit vectors meaning $Q^T Q = QQ^T = I$) and R is an upper triangular matrix. If A is invertible, then the factorization is unique if we require the diagonal elements of R to be positive.

If instead A is a complex square matrix, then there is a decomposition $A = QR$ where Q is a unitary matrix (so $Q^*Q = QQ^* = I$).

If A has n linearly independent columns, then the first n columns of Q form an orthonormal basis for the column space of A. More generally, the first k columns of Q form an orthonormal basis for the span of the first k columns of A for any $1 \leq k \leq n$. The fact that any column k of A only depends on the first k columns of Q is responsible for the triangular form of R.

More generally, we can factor a complex $m \times n$ matrix A, with $m \geq n$, as the product of an $m \times m$ unitary matrix Q and an $m \times n$ upper triangular matrix R. As the bottom $(m - n)$ rows of an $m \times n$ upper triangular matrix consist entirely of zeroes, it is often useful to partition R, or both R and Q:

$$A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1, Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1,$$

where R_1 is an $n \times n$ upper triangular matrix, 0 is an $(m - n) \times n$ zero matrix, Q_1 is $m \times n$, Q_2 is $m \times (m - n)$, and Q_1 and Q_2 both have orthogonal columns.

If A is of full rank n and we require that the diagonal elements of R_1 are positive then R_1 and Q_1 are unique, but in general Q_2 is not. R_1 is then equal to the upper triangular factor of the Cholesky decomposition of A^*A ($= A^T A$ if A is real).

3 Algorithms

3.1 Cholesky-QR

To orthogonalize a tall-skinny matrix, the Cholesky QR (CholQR) requires only one global reduction between the parallel processing units, while performing most of its computation using BLAS-3 kernels. Hence, compared to other orthogonalization schemes, it obtains exceptional performance on many of the modern computers. However, the orthogonality error of CholQR is bounded by the squared condition number of the input matrix, and it is numerically unstable for an ill-conditioned input matrix.

Algorithm 1 Cholesky-QR

1: **function** CholQR(V)
   ▷ *Gram matrix computation*
2: $B = V' * V$
   ▷ *Cholesky*
3: $R = chol(B)$
   ▷ *Orthonormalize V*
4: $Q = V * R^{-1}$
5: **return** $[Q, R]$

3.2 Classical Gram-Schmidt

Gram-Schmidt (GS) is the most well known and also the least computationally expensive method for QR factorization. In addition, this method is based primarily on level 2 BLAS kernels, and it is possible to implement it in parallel with a modest number of $m + 1$ synchronization points. Although computationally attractive, GS not always produce numerically orthogonal vectors. However, when used with reorthogonalization, a second GS iteration is typically enough for producing orthogonality to machine precision. Since a second iteration is not always necessary, GS with reorthogonalization is often preferred over other methods.

Algorithm 2 Classical Gram-Schmidt

1: **function** CGS(V)
2: **for** j = 1, 2, ... n **do**
3: $r(1 : j-1, j) = Q^T(:, 1 : j-1)' * V(:, j)$
4: $v = V(:, j) - Q(:, 1 : j-1) * R(1 : j-1, j)$
5: $R(j, j) = norm(v)$
6: $Q(:, j) = v/R(j, j)$
7: $R(:, m+1 : n) = Q' * V(:, m+1 : n)$
8: **return** [Q, R]

3.3 Modified Gram-Schmidt

Modified Gram-Schmidt (MGS) improves the numerical stability of the GS by orthogonalizing individual pairs of vectors rather than a vector against a block. The operation count for MGS is the same with GS, but working on individual vectors allows only for level 1 BLAS kernels, which impairs significantly the actual performance on cache based computers. Finally, the number of synchronization points grows quadratically with m.

Algorithm 3 Modified Gram-Schmidt

1: **function** MGS(V)
2: **for** i = 1 to n **do**
3: **for** j = 1 to i-1 **do**
4: $R(j,i) = Q^T(:,j)' * V(:,i)$
5: $V(:,i) = V(:,i) - R(j,i) * Q(:,j)$
6: $R(i,i) = norm(V(:,i))$
7: $Q(:,i) = V(:,i)/R(i,i)$
8: $R(:,m+1:n) = Q' * V(:,m+1:n)$
9: **return** [Q, R]

3.4 Singular value QR

There are various ways to obtain an orthonormal basis of the $span(W)$ rather than a QR factorization, an especially interesting one is the one derived from the right singular vectors of W. This method computes the upper-triangular matrix by first computing the singular value decomposition (SVD) of the Gram matrix, $U\Sigma U^T := B$, followed by the QR factorization of $\Sigma^{1/2}U^T$. Then the column vectors are othonormalized through the triangular solve $Q := VR^{-1}$. Compared to Cholesky factorization the SVD and QR factorization of the Gram matrix is computationally more expensive. However, the dimension of the Gram matrix is much smaller than that of the input matrix V.

Algorithm 4 Singular value QR

1: **function** SVQR(V)
2: $B = V' * V$
3: $[V, \Sigma, U] = svd(B)$
4: $[V, R] = qr(\sqrt{\Sigma}U^T)$
5: $Q = V * R^{-1}$
6: **return** [Q, R]

3.5 Communication-avoiding QR

Communication-avoiding QR orthogonalizes the set of n vectors V against each other through a tree reduction of the local QR factorization. Like CholQR, to orthogonalize V, CAQR requires only one global reduce.

Algorithm 5 Communication-Avoiding QR

1: **function** CAQR(V)
2: $[X, R] = qr(V)$
3: $[Y, R] = qr(R)$
4: $Q = X * Y$
5: **return** [Q, R]

4 Results of the study

To test the performance and the behavior of the various methods, we decided to test them on the same machine to have a direct comparison of the computing time. As the computing time is relative to the hardware requirements, we completed the test in a relaxed environment as neither of the running processes could influence the results.

For testing the GPU we used the data structures provided by the MatLAB Parallel Processing Toolbox add-on. The data structure is *gpuarray*, an object constructed on the CUDA framework that has implemented the majority of MatLAB functions, such as algebraic operations and matrix operations. The native function used on the project were all supported by this object.

We also noticed that this function is not well implemented or it has some security restrictions that limited the laptop test on smaller matrices that the GPU actually could support. In fact, the GPU at max was used for ~860 MB instead of the whole 1024 MB available.

For compute a valid running time we used the native functions *timeit* for CPU runs and *gputimeit* for GPU runs. These functions call the specified method multiple times, and compute the median of the measurements.

Alongside the methods presented in Section 3 we also tested the native QR MatLAB function expecting somehow that the run results were in the fastest methods, assuming that the function was more optimized on a low level programming.

We run the tests with 200 random generated matrices with dimension $50 \times i$ rows and 30 columns. We chose a fixed number of columns as the original article worked on that number and the complexity and number of operations would not have grown drastically.

4.1 Intel i7 - NVidia GeForce 640M LE test

The first test was performed on a laptop with processor Intel i7-3520M turbo boosted at 3.40 GHz, 8 GB of RAM and graphic card NVidia GeForce GT 640M LE 1 GB (CUDA compute capability of 3.0). The laptop had got another GPU, an Intel HD 4000. This card was not used for the test as it has not CUDA support but the whole system runs on it, so the NVidia card was exclusively used only for the test.

Although, the graphic card was not able to perform all the tests, finishing at the round #112. In the round #113 the program exited because the GPU memory was saturated using ~860 MB of the 1024 MB available. It reached memory overflow on CAQR GPU computation (Section 3.5), there were present in memory four gpuarrays, matrix V 5650x30 elements, matrix R 5650x30 elements, matrix X 5650x5650 elements, matrix Y 5650x5650 elements and the program was trying to execute the final method operation, the matrix multiplication $X \times Y$. To avoid the problem we tried to break in cycles the multiplication but in this way the algorithm was too slower than the direct multiplication, we tried using a sequential approach or even a multithread approach. The multithread approach was very delicate to perform because we were doing it for a memory problem and we had to be very careful on how many concurrent threads to run.

However the results were quite interesting, so we decided to analyze them to the #112 round, from an input matrix of 50x30 elements to one 5600x50 elements.

In Figure 1 we can find the results of the test. We can see how CholQR is the fastest algorithm between CPU and GPU. Differently to what we expected, the GPU tests were slower than the CPU respective. However, we see also that the computation on the GPU, apart of CAQR and native QR, has a running time stable instead of grow like CPU algorithms. We can hypothesize that if we would have very big matrices, the GPU computation of CholQR, CGS, MGS and SVQR could be fastest than their respective algorithm run on CPU. For instance, CholQR starts to run in $\sim 10^{-4}$ seconds to finish in around $\sim 2 * 10^{-3}$ seconds (+1900%) and the GPU time starts with $\sim 3 * 10^{-3}$ seconds, increasing from the start only by $\sim 10^{-3}$ seconds (+140%).

On Table 1 we can find a briefly representation of the computation differences between the first and last test on each algorithm. The results on the table have to be taken very carefully as it is the difference between only two measures and we could induce a non indifferent error. However, we can take the percentages as a relation between themselves, and be quite confident to say that the difference between the computational time grow of the GPU runs is much smaller than the CPU one. Assumed this, we can also hypothesize that in relation to the speed and the percentage difference, the SVQR algorithm can become the fastest algorithm from a still unknown very large matrix.

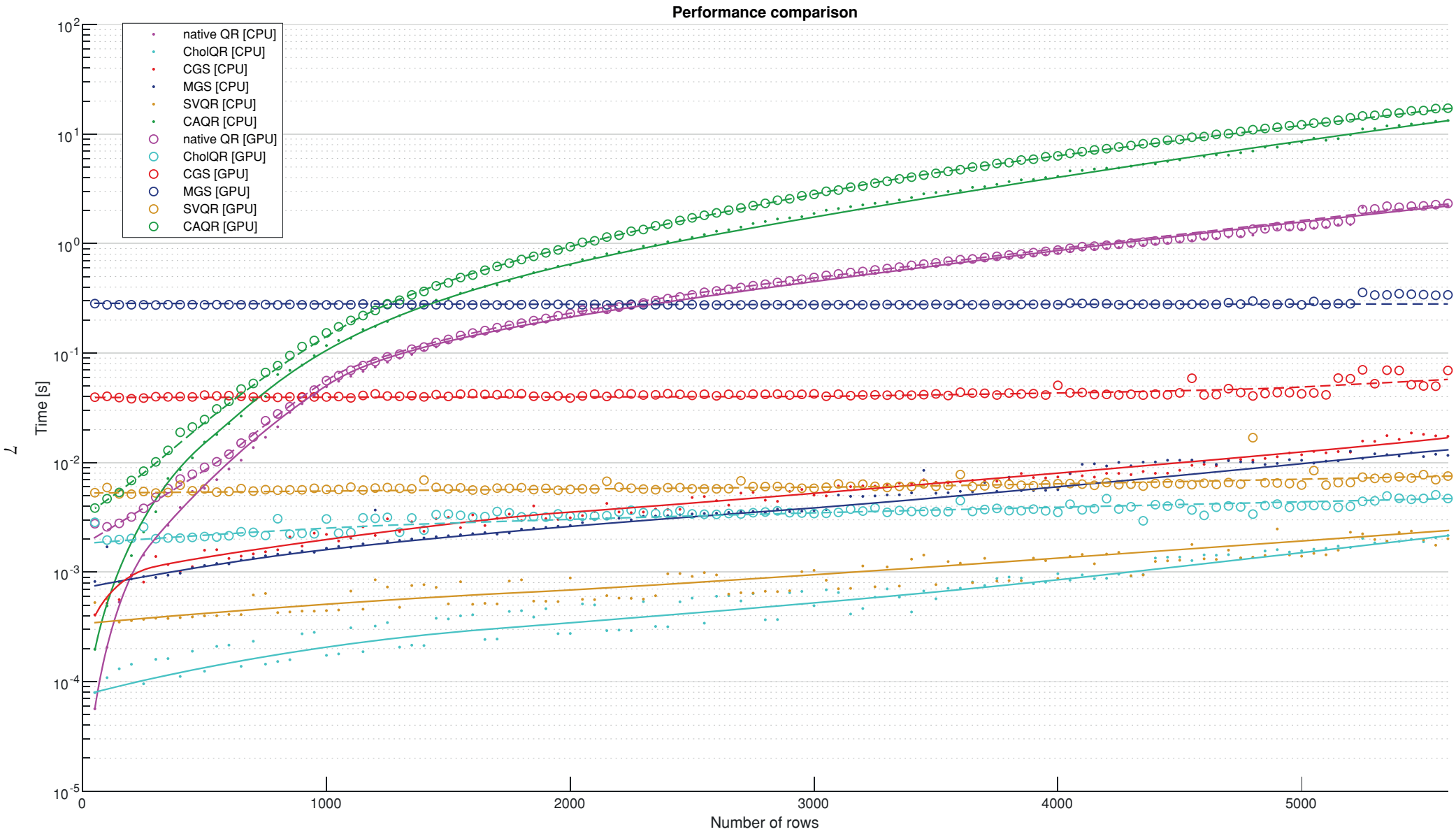

Figure 1: Test on Intel i7 - NVidia GeForce 640M LE

	Difference time #1 - #112 round	
	CPU	GPU
native QR	1,07E+06%	8,95E+04%
CholQR	1889%	140%
CGS	3245%	72%
MGS	584%	23%
SVQR	255%	27%
CAQR	2,70E+06%	3,68E+05%

Table 1: Running time difference between #1 and #112 round

4.2 Intel i5 - NVidia GeForce 1070 test

The final test was performed on a desktop computer with processor Intel i5-7600 at 3.50 GHz, 16 GB of RAM and graphic card NVidia GeForce GTX 1070 6 GB (CUDA compute capability of 6.1). On this machine we could not have the total control as it was borrowed and we could only run the initial planned 200 rounds. The results are as well interesting but they could be more if it was possible to run more tests with bigger matrices.

In Figure 2 we can find the results of the test. We can see how the behavior of the chart is similar to the laptop one. CholQR is still the fastest algorithm on CPU and GPU. Also, in this chart the behavior of the CGS algorithm is quite interesting. The CPU runs are all faster than GPU ones but around the #200 round with a input matrix size of 10000x30 the speed time of the CPU is nearly the same to the GPU one and we think that with a bigger matrix the CPU run would be slower than the GPU.

As before, CholQR, SVQR, CGS and MGS have a running time very stable in respect of the CPU time grow and Table 2 can confirm it. CholQR on GPU increases only of 43% versus the +1990% of the CPU (at round #112 for direct confrontation with laptop GPU is +10% and CPU +1085%) and SVQR has an increase of 11% on GPU and 392% on CPU (#112 round GPU +0.57% and CPU +174%). Like the laptop's test, we can hypothesize that in relation to the speed and the time grow, the SVQR algorithm can become the fastest algorithm from a still unknown very large matrix.

	Difference time #1 - #112 round		Difference time #1 - #200 round	
	CPU	GPU	CPU	GPU
native QR	1,10E+06%	4,27E+04%	3,41E+06%	1,33E+05%
CholQR	1085%	10%	1990%	43%
CGS	3411%	-1,61%	9874%	12%
MGS	1330%	-4,07%	2008%	2,09%
SVQR	174%	0,57%	392%	11%
CAQR	1,73E+06%	5,88E+04%	8,58E+06%	2,70E+05%

Table 2: Running time difference between #1 and #112 round and between #1 and #200 round

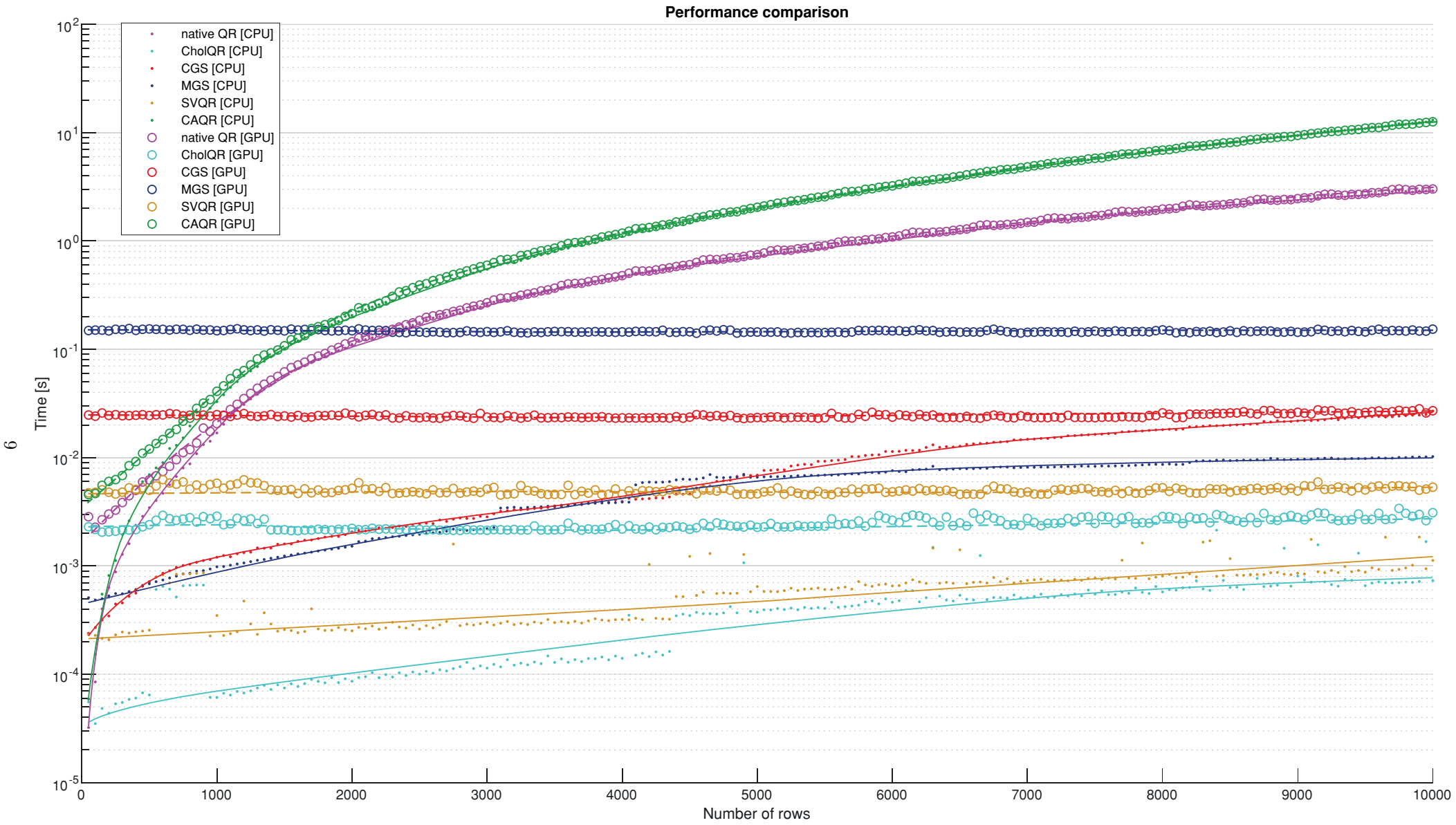

Figure 2: Test on Intel i5 - NVidia GeForce 1070

5 Conclusions

We saw from both the tests executed how Cholesky-QR is a very fast method of factorization on small to medium matrices. The test's results were surprising.

In the tests we speculated how Single Value QR method could become the fastest algorithm using very large matrices. A needed extension of the work is to prove this statement with a very powerful graphic card. In general we could use of a more powerful hardware with total control on it to test all the methods on large matrices. The improvement on the hardware side could be done vertically using a more powerful graphic card and/or horizontally using more than one GPU running concurrently.

A future work would require a smarter selection of the programming language used, MatLAB has the pro to be math-friendly but with its native structures it seems it is not suitable for efficient programming. We could suggest the use of C or C++ using CUDA libraries. This will allow the access to a lower programming level and gain more control of data structures and basic operations.

The need to prove the algorithms on larger matrices derives from the expectations that we had before starting the tests. We expected that a graphic card with much more cores than a CPU (384 cores on NVidia GeForce 640M LE and 1920 cores on NVidia GeForce GTX 1070) could be faster at computing parallel operations. Instead, we found that the CPU respective algorithms were as fast as, or faster than the GPU we used. We also expected that the native QR function would be one of the fastest methods, however it appeared to be one of the slowest.

References

[1] I. Yamazaki, S. Tomov, J. Dongarra, *Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multipleGPUs*, SIAM J. Sci. Comput., 37 (2015), pp. 307–330

[2] A. Stathopoulos, K. Wu, *A block orthogonalization procedure with constant synchronization requirements*, SIAM J. Sci. Comput., 23 (2002), pp. 2165-2182

[3] G. H. Golub, C. F. Van Loan, *Matrix computations*, The Johns Hopkins University Press, Baltimore and London, 3rd edition, 1996

[4] R. A. Horn, C. R. Johnson, *Matrix Analysis*, Cambridge University Press, 2nd edition, 2012

[5] S. J. Julier, J. K. Uhlmann, *A General Method for Approximating Nonlinear Transformations of ProbabilityDistributions*, University of Oxford, 1996

[6] *gpuArray*, MatLAB documentation, `https://it.mathworks.com/help/distcomp/gpuarray_object.html`

[7] *CUDA GPUs*, `https://developer.nvidia.com/cuda-gpus`

[8] *NVidia GeForce GT 640M LE Specifications*, `https://www.geforce.com/hardware/notebook-gpus/geforce-gt-640m-le/specifications`

[9] *NVidia GeForce GTX 1070 Family*, `https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1070-ti`