Exploring QR factorization on GPU for Quantum Monte Carlo Simulation

Benjamin McDaniel, Ming Wong

August 7, 2015

Abstract

QMCPACK is open-source scientific software designed to perform Quantum Monte Carlo simulations, which are first principles methods for determining the properties of the electronic structure of atoms, molecules, and solids. One major objective is finding the ground state for a physical system. We will investigate possible alternatives to the existing method in QMCPACK such as QR factorization for evaluating single-particle updates to a system's electron configuration by improving the computational efficiency and numerical stability of the algorithms.

1 Introduction

Quantum Monte Carlo (QMC) simulations can be used to accurately describe physical systems (Ceperly/Chester 3081). These simulations use Slater-Jastrow trial wave functions to evaluate possible changes to the system, one particle at a time, for their potential effect on the overall energy of the system (Nukala/Kent 130).

The Slater determinant factor of the trial wave function for the entire system of n electrons can be represented as

$$\Psi(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \chi_1(\mathbf{x}_1) & \chi_2(\mathbf{x}_1) & \cdots & \chi_N(\mathbf{x}_1) \\ \chi_1(\mathbf{x}_2) & \chi_2(\mathbf{x}_2) & \cdots & \chi_N(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \chi_1(\mathbf{x}_N) & \chi_2(\mathbf{x}_N) & \cdots & \chi_N(\mathbf{x}_N) \end{vmatrix}$$

where $\chi_{1,2\cdots n}$ represents the single-particle wave functions: $\chi_(\mathbf{r}_i) = \sum_{j=1}^{n} \phi_j \mathbf{c}_j \mathbf{r}_i$.

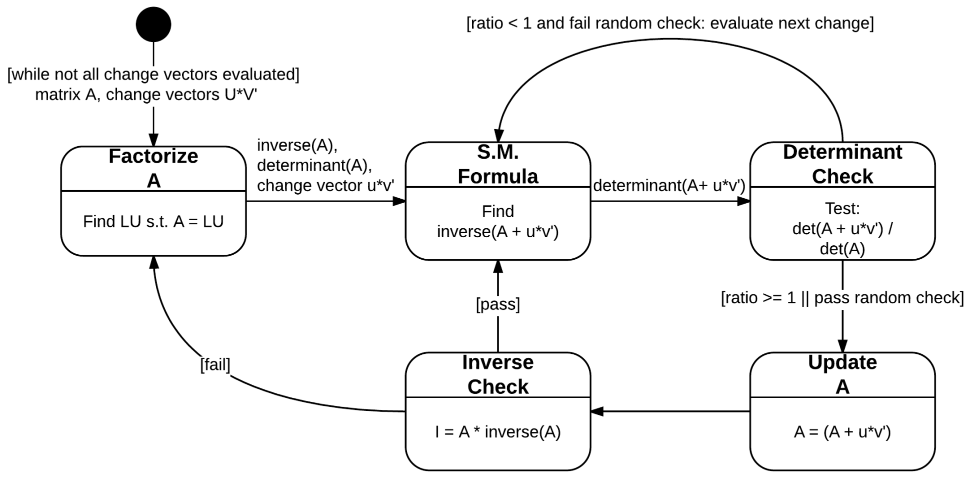
A single particle's contribution to the overall energy of the system is therefore encoded in a single row of the above matrix, and the effect of a proposed single-particle position change on the overall energy of the system can be assessed by altering a single row of the matrix with the proposed change, and re-computing the determinant. The ratio of the resulting new determinant to the existing determinant, $\dfrac{det(A')}{det(A)}$, determines whether or not the change is accepted. The proposed change is accepted if the ratio is greater than 1. If the ratio is less than 1, it is compared to a randomly generated value x from 0 to 1, and accepted only if the ratio is greater than x. This additional acceptance criterion prevents "stagnant" simulation states.

2 Background

The current implementation in QMCPACK for this procedure uses LU decomposition on the above matrix, matrix A, yielding lower and upper triangular factors from which the inverse and determinant of A can be easily found. If a proposed single change to a column k is represented as $A + ue'_k$ where ue'_k is the outer product of column vector u (the desired new values for $A(:,k)$ minus the old values of $A(:,k)$) and standard basis vector e'_k, the Sherman-Morrison formula can be employed to compute the inverse of A corrected for the change:

$$(\mathbf{A} + \mathbf{u}\mathbf{e}'_\mathbf{k})^{-\mathbf{1}} = \mathbf{A}^{-\mathbf{1}} - \frac{\mathbf{A}^{-\mathbf{1}}\mathbf{u}\mathbf{e}'_\mathbf{k}}{\mathbf{1} + \mathbf{e}'_\mathbf{k}\mathbf{A}^{-\mathbf{1}}\mathbf{u}}$$

With this updated inverse, and the existing cofactor of A, the determinant as modified by the single column change is computed. If the change is accepted after the evaluation of the ratio of modified determinant to existing determinant as described in Section 1, the addition $\mathbf{A} = \mathbf{A} + \mathbf{u}\mathbf{e}'_\mathbf{k}$ is performed. By repeating this process for all proposed column changes, each can be evaluated, accepted/rejected, and added to the system represented by Aif appropriate.



[ratio < 1 and fail random check: evaluate next change]
[while not all change vectors evaluated]
matrix A, change vectors U*V'
Factorize A
Find LU s.t. A = LU
inverse(A), determinant(A), change vector u*v'
S.M. Formula
Find inverse(A + u*v')
determinant(A+ u*v')
Determinant Check
Test: det(A + u*v') / det(A)
[ratio >= 1 || pass random check]
[pass]
[fail]
Inverse Check
I = A * inverse(A)
Update A
A = (A + u*v')

Figure 1: A flowchart for the existing QMCPACK single column update.

Unfortunately, LU decomposition is not always numerically stable when pivoting is employed, especially for certain pathological value patterns in A. In other words, the inverse of A after computing the LU decomposition, such that $\mathbf{A_{cal}^{-1}} = \mathbf{LUP^{-1}}$ can be multiplied to the original A_o to obtain $\mathbf{A_o A_{cal}^{-1}}$, is not always precisely equal to the true A^{-1} defined by $\mathbf{A_o A^{-1}} = \mathbf{I}$.

Additionally, after several single column updates using the Sherman-Morrison formula, it is possible that the computed A^{-1} will diverge from the mathematical A^{-1}, requiring another LU decomposition be performed on A to update A^{-1}.

3 Concept

We will examine possible improvements that could be made to the current approach within the context of QMC simulations.

3.1 Improvements related to QR factorization

The proposed approach includes computing and utilizing the QR factorization of A, $QR = A$, where Q is an orthogonal matrix and R is upper triangular, as opposed to the current LU factorization. Even though both LU and QR factorization utilize $O(n^3)$ operations, QR factorization is often less sensitive to ill-conditioned matrices than LU decomposition, and can provide greater relative stability after changes are made. This obviates the need for the QMCPACK to periodically recompute the explicit inverse of A.

In addition, the change vector u can be applied to the right side of the relationship $\mathbf{A} = \mathbf{QR}$:

$$(\mathbf{A} + \mathbf{ue'_k})^{-\mathbf{1}} = \mathbf{QR} + \mathbf{ue'_k} = \mathbf{Q}(\mathbf{R} + \mathbf{Q'ue'_k})$$

Because Q is orthogonal, $det(Q) = \pm 1$ and $det(R) = \pm det(A)$ we can determine how a column change to A will affect the ratio of $\dfrac{det(A')}{det(A)}$ by evaluating $\dfrac{\mathbf{det}(\mathbf{R} + \mathbf{Q'ue'_k})}{\mathbf{det}(\mathbf{R})}$.

If the proposed change is accepted, then A can be updated simply by performing the addition $\mathbf{R} = \mathbf{R} + \mathbf{Q'ue'_k}$. R must be returned to upper triangular form at this point, but this process can be performed with great efficiency by utilizing some known features of R, as we will demonstrate in Section IV. If, instead, the addition is delayed, we can also limit the need to re-triangularlize R.

3.2 Triangular Solve

In our proposed scheme, the vector w to update R is calculated by the product of the transpose of Q and the updated column vector u. Unlike the previous approach that involves the Sherman Morrison formula to explicitly calculate inverse of A, we proposed to simply use the triangular solve between the transpose of R and $e'_k$to obtain a row of modified R. Then this row of the inverse of modified R can be plugged into a matrix determinant lemma:

$$\mathbf{det}(\mathbf{R} + \mathbf{WV'}) = \mathbf{det}(\mathbf{I_m} + \mathbf{V'R^{-1}U})\mathbf{det}(\mathbf{R})$$

to estimate the determinant of R. As with the existing approach, this estimated determinant can be utilized later to determine the feasibility of the proposed column update.

3.3 Rank-k updates

To limit the need to re-triangularize R, contiguous accepted change columns, $\mathbf{Q'ue'_k}$ can be stored in the columns of a submatrix W, but not yet added to R. Then, the estimated determinant for a series of changes can still be found using the Matrix Determinant Lemma for the rank-k case by using the following:

$$\mathbf{det}(\mathbf{R} + \mathbf{WV'}) = \mathbf{det}(\mathbf{1} + \mathbf{v'R^{-1}u})\mathbf{det}(\mathbf{R})$$

where for previously accepted (but not added) changes $i...n-1$, and change being evaluated n, V' is the zero matrix matching A^{-1} in size, but with transposed standard basis vectors $e_k(i...n)$ in the i to n columns of V'.

Observe that the product $V'R^{-1}U$ is a small matrix relative to R, whose determinant with I_m can be quickly computed, especially if certain helpful techniques are employed. Multiple column changes can thus be evaluated, and then added to R en bloc, delaying the process of returning R to upper triangular form.

4 Implementation

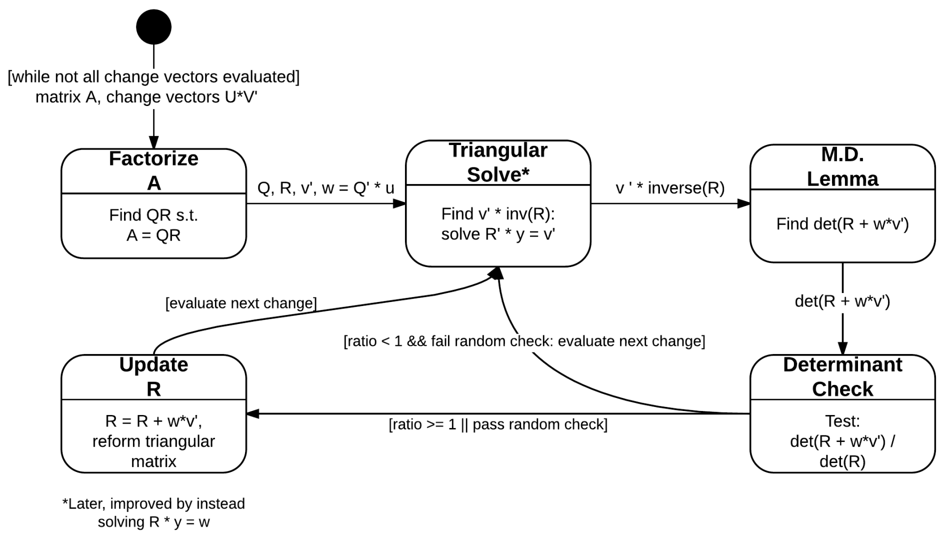

[while not all change vectors evaluated]
matrix A, change vectors U*V'
Factorize A
Find QR s.t. A = QR
Q, R, v', w = Q' * u
Triangular Solve*
Find v' * inv(R): solve R' * y = v'
v ' * inverse(R)
M.D. Lemma
Find det(R + w*v')
det(R + w*v')
Determinant Check
Test: det(R + w*v') / det(R)
[ratio < 1 && fail random check: evaluate next change]
[ratio >= 1 || pass random check]
Update R
R = R + w*v', reform triangular matrix
[evaluate next change]
*Later, improved by instead solving R * y = w

Figure 2: A flowchart for the existing QMCPACK single column update.

The flowchart from Figure 2 presents the algorithm for the rank-1 update. This algorithm has been implemented using CUDA and tested on a GPU equipped node of the Beacon cluster at the National Institute for Computational Sciences at Oak Ridge. The GPU accelerator is a Tesla K20Xm with a GK110 processor. The program begins with driver that invokes two primary kernels to perform rank-1 updates to a test set of randomly generated floating-point matrices.

First, for each test matrix A, QR decomposition is performed. As mentioned in section 2, since Q is orthonormal, the absolute value of the determinant of A can be calculated using the determinant of the upper triangular matrix R. A set of u vectors, the proposed column changes to A to be evaluated, are randomly generated in the program. Once matrices Q and R and vectors u have been determined, the estimate delta kernel is invoked first. Inside the estimate delta kernel, vector w contains the update as applied to R, calculated as the product of Q' and u. It also utilizes a triangular solve child kernel to determine the delta required for the determinant check.

If the proposed column update passes the determinant check, we will proceed to update R immediately by calling the update QR kernel. One important point to highlight is that unlike the existing approach, there is no longer an inverse check to ensure that the inverse of A is still mathematically correct since we do not require calculating any matrix inverse.

This updated column k of R will be logically permuted to the end of matrix, while the columns to the right of the k^{th} column will be shifted left one position, although no columns of R are ever moved in physical memory. In this way, the updated R is upper Hessenberg, and a permutation vector maps the logical column locations to the physical column locations. Givens rotation matrices are used to annihilate non-zero entries below the diagonal as shown in Figure 3. These operations are also applied to matrix Q to maintain the $A = QR$ relationship. Using these methods, a series of updates can be evaluated and applied for each test matrix.

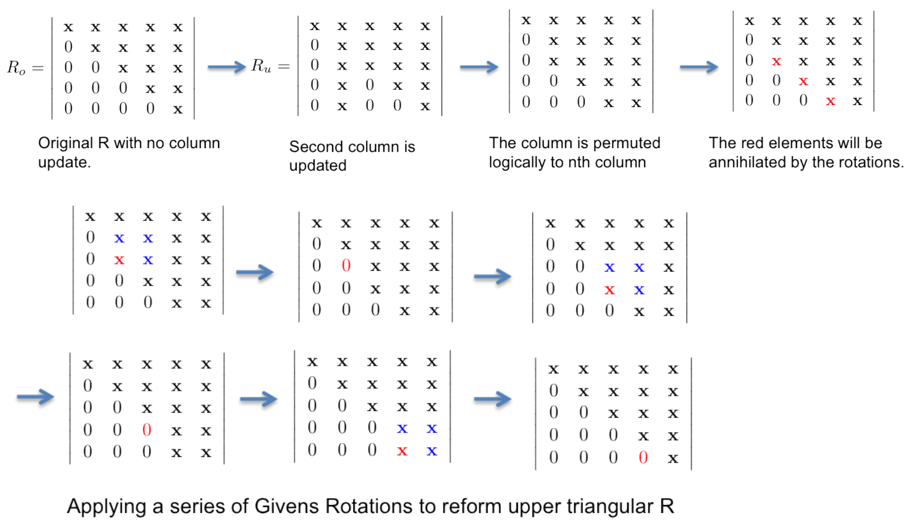

Original R with no column update.
Second column is updated
The column is permuted logically to nth column
The red elements will be annihilated by the rotations.
Applying a series of Givens Rotations to reform upper triangular R

Figure 3: Applying Givens Rotations.

5 Results

5.1 Findings

The estimated memory footprint (GPU DRAM) for this algorithm is: $(2n^2 + 2n)$ * sizeof(⟨floating point datatype used⟩) per matrix, where n is the size of the input matrix A.

For a single execution of both kernels, about $15n^2$ flops are required per matrix. This figure is determined by summing the cost of one matrix vector multiplication, a triangular solve with one right hand side, a matrix vector addition and from 1 to n iterations of finding and applying Givens rotations. We measured the flops executed by our kernels using the Nvidia Visual Profiler and found that the flop cost in practice is extremely close to our $15n^2$ estimate.

The efficiency of the algorithm measured in column updates per second (both evaluation and application) for different input matrix sizes and number of test matrices is displayed below.

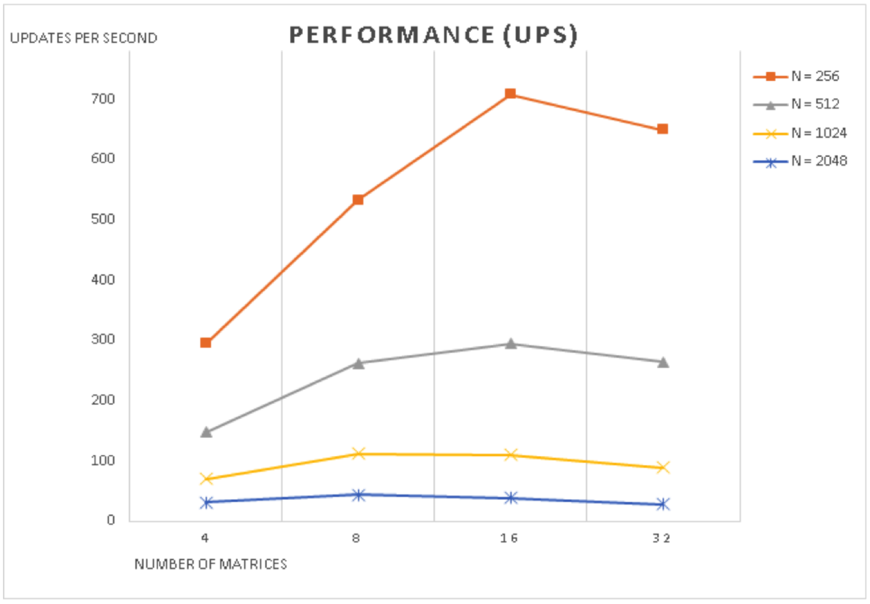


Figure 4: Updates Per Second Graph.

The most significant trend demonstrated is the "flattening" of parallelization gains as the matrix size increases. Since we are well within the computational capability of the GPU accelerator, we can be sure that control flow or memory latency issues exist within our current implementation.

5.2 Analysis

Two potential avenues for improvement were identified.

1) We observe that the transformations used to return R to upper triangular form, Givens rotations, may not be ideal in our specific case (annihilating a single subdiagonal nonzero entry in several columns of a dense, square matrix). While our algorithm uses a sequential Givens algorithm, it is not clear that existing parallel Givens QR algorithms, many of which are based on the Sameh and Kuck scheme (Sameh and Kuck, 1978) and exploit the need to annihilate multiple nonzero entries in each column, will provide much increased efficiency.

Instead, we believe that using Householder reflectors may be more helpful. The flop cost of this technique may be greater in our case, outlined above, but many of these flops will be contained in a highly optimized Level 3 BLAS call (GEMM).

2) The use of matrix permutations, requiring a vector to map logical columns to their physical (in memory) locations, is another source of control flow complexity.

A well-known technique to make a rank-1 column update to an upper triangular matrix, resulting in an upper Hessenburg matrix, could be helpful (Golub and Van Loan, 1996, p. 607). Using this method, we could avoid maintaining the permutation vector, and obviate the need to use custom kernels/functions that accomodate the permutations.

In addition to these changes, better aligning our implementation with CUDA/ cuBLAS best practices could provide a significant increase in efficiency.

6 Conclusion

We still need to evaluate our method with the existing procedure for a meaningful comparison of numerical stability and execution efficiency. Our work to this point indicates that using the QR factorization may be an effective method for proposing and apply rank-1 updates within QMCPACK.

7 References

Andrew, Robert, and Nicholas Dingle. ”Implementing QR Factorization Updating Algorithms on GPUs.” Parallel Computing 40.7 (2014): 161-72. Web. 4 Aug. 2015. ¡http://www.sciencedirect.com/science/article/pii/S0167819114000337¿.

“CuBLAS :: CUDA Toolkit Documentation.” CuBLAS :: CUDA Toolkit Documentation. Web. 4 Aug. 2015.
newline

Golub, Gene H., and Charles F. Loan. Matrix Computations. 3rd ed. Baltimore: Johns Hopkins UP, 1996. Print.

Kontoghiorghes, Erricos J. ”Parallel Strategies for Rank- K Updating of the QR Decomposition.” SIAM. J. Matrix Anal. and Appl. SIAM Journal on Matrix Analysis and Applications 23.3 (2000): 714-25. Web. 4 Aug. 2015. ¡http://epubs.siam.org/doi/pdf/10.1137/S0895479896308585¿.

Kontoghiorghes, Erricos John. ”Greedy Givens Algorithms for Computing the Rank-k Updating of the QR Decomposition.” Parallel Computing 28 (2002): 1257-273. Web. 4 Aug. 2015. ¡http://www.dcs.bbk.ac.uk/ matrix/Papers/ErricosRankk.pdf¿.

Padua, David A. Encyclopedia of Parallel Computing. Vol. 4. New York: Springer, 2011. Print.

Sameh, A. H., and D. J. Kuck. ”On Stable Parallel Linear System Solvers.” Journal of the ACM JACM J. ACM 25.1 (1978): 81-91. Web. 4 Aug. 2015. ¡http://dl.acm.org/citation.cfm?id=322054¿.

Volkov, V., and J.w. Demmel. ”Benchmarking GPUs to Tune Dense Linear Algebra.” 2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (2008). Web. 4 Aug. 2015. ¡http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf¿.