

Sherman-Morrison-Woodbury kernels for QMC and other applications

François Coppens* Pablo Oliveira † Eric Petit ‡

December 15, 2022

Contents

1 Aims and contributions	3
2 Introduction	3
2.1 The case for using Sherman-Morrison in QMC=Chem	3
2.2 The Sherman-Morrison formula	4
2.3 The iterative problem of SM in QMC=Chem	5
3 Numerical methods	6
3.1 An naïve approach to iterative Sherman-Morrison	6
3.2 Partial pivoting: Maponi’s method for solving linear systems	7
3.3 Reordering the updates	8
3.4 Slagel’s method of update splitting	9
3.5 All at once: the Woodbury matrix identity	10
4 HPC versions	11
4.1 Vectorisation	11
4.1.1 Zero-padding of arrays	11
4.1.2 Pointer aliasing	12
4.1.3 Contiguous memory and memory alignment	12
4.2 Manual loop-unrolling: 2×2 and 3×3 WB and the Blocking kernel	12
4.3 Explicit array dimensions and kernel-template generation	14
5 Experiments with QMC=Chem	15
5.1 Dataset extraction	15
5.2 Problems in the current implementation	16
5.2.1 Issue 1: Numerical accuracy with large number of determinants	16
5.2.2 Issue 2: Low arithmetic intensity (HPC)	16
5.3 Measurement machine architecture and measurement procedure	17
5.4 Results for the numerical accuracy	18
5.4.1 329 α -determinants	19
5.4.2 15784 α -determinants	19
5.5 Results for the performance	21
5.5.1 Performance per rank-1 update	21
5.5.2 Performance stratification	22

*UVSQ

†UVSQ

‡Intel Corp.

6 Classification of all the kernels	25
6.1 Final recommendations	25
7 Inclusion in Quantum Monte Carlo Kernel Library	26
8 Preliminary work on $K \times K$ Woodbury on the GPU	26
8.1 GPU warm-up effects	26
8.2 Multi-threaded CPU vs GPU	27
Appendix A Zero-padding	29
A.1 Sherman-Morrison with zero-padding	29
A.2 Woodbury with zero-padding	30
References	30

1 Aims and contributions

This work has been executed in the context of the TREX-CoE[3]. To that end we aim to address the following goals. To

- solve a numerical problem that exists in the Sherman-Morrison method, as implemented and used extensively in the QMC=Chem[7] software package,
- extend and generalise this method to include higher-rank cases (Woodbury matrix identity),
- implement these methods and optimise them to run efficiently on high performance computing architectures.

QMC=Chem is used as a test case to address the numerical problems in its implementation of the Sherman-Morrison method. However, it is often used in other Quantum Monte Carlo applications as well. Indeed, the method itself is more general and can be employed in many other application domains. Because of this, these algorithms have been made available to the public in the form of computational kernels inside the Quantum Monte Carlo Kernel Library[2].

Finally, we include performance- and numerical benchmarks to demonstrate that the numerical issues that have been identified in QMC=Chem have been resolved and the stated objectives have been met.

2 Introduction

As was mentioned before, the focus of this work addresses the particular numerical problems of the implementation of the Sherman-Morrison method in the TREX-CoE code ‘QMC=Chem’.

Even though the work and results presented and discussed here are obtained with QMC=Chem, the final kernels that have been included in Quantum Monte Carlo Kernel Library (QMCkl) can be used in other Quantum Monte Carlo (QMC) applications that have a need for fast updating an inverse matrix.

In section 2.1 we briefly explain why the Sherman-Morrison method is used in QMC=Chem. Then we formally introduce the Sherman-Morrison formula in Section 2.2. Finally, in Section 2.3 the problem with the current iterative implementation and how it arises will be explained as well.

2.1 The case for using Sherman-Morrison in QMC=Chem

In many QMC methods the many-body wave function $\Psi(\mathbf{r}_1, \dots, \mathbf{r}_N)$ for N electrons (where \mathbf{r}_i is the location of electron $i \in \{1, \dots, N\}$) is expressed as an expansion of N_{det} determinants of Slater-matrices, containing ordered collections of single-electron basis-wave functions

$$\Psi(\mathbf{R}) = \sum_{k=1}^{N_{\text{det}}} c_k \det S_k^{\uparrow}(\mathbf{R}_{\uparrow}) \det S_k^{\downarrow}(\mathbf{R}_{\downarrow}) \quad (1)$$

where $\mathbf{R} = \mathbf{r}_1, \dots, \mathbf{r}_N$, \mathbf{R}_{\uparrow} and \mathbf{R}_{\downarrow} the subsets of coordinates associated with spin-up and spin-down electrons. The matrix elements of S_k^{σ} ($\sigma = \uparrow, \downarrow$) are defined as

$$[S_k^{\sigma}]_{ij} = \phi_j(\mathbf{r}_i) \quad (2)$$

and ϕ_j are the single-electron orbitals. The coefficients c_k take care of the normalisation of the many-body wave function ($\int_{-\infty}^{+\infty} |\Psi|^2 d\mathbf{R} = 1$).

Sometimes this expansion only contains one big Slater determinant, sometimes it contains many smaller ones. In our test-case, based on data extracted from QMC=Chem, it concerns the latter.

During the course of minimising the total energy of the system, in each Monte Carlo step the kinetic energy of the system

$$E_{\text{kin}} \propto -\sum_{i=1}^N \nabla_{\mathbf{r}_i}^2 \Psi(\mathbf{r}_1, \dots, \mathbf{r}_N) \quad (3)$$

needs to be calculated. Each Laplacian $\nabla_{\mathbf{r}_i}^2$ is acting only on the $\phi_j(\mathbf{r}_i)$ parts of the wave function. To compute $\nabla_{\mathbf{r}_i}^2 \Psi$ for a given \mathbf{r}_i the following identity is used extensively

$$\partial_{r_i} \det S = \det S \text{Tr}(S^{-1} \partial_{r_i} S), \quad r_i \in \{x_i, y_i, z_i\} \quad (4)$$

We therefore need to keep track of the inverses of the Slater-matrices to compute basic quantities like the kinetic energy of the system.

Common manipulations of the many-body system to move towards a lower total energy are moving single electrons and/or manipulating electron orbitals. This corresponds to changes either in the rows or the columns in the Slater-matrix. Whenever there is a change in the Slater-matrix due to one these manipulations, the inverse of the Slater-matrix needs to be re-computed.

A full inverse can be computed with software libraries like LAPACK[4] or MKL[1], but it turns out that we do not need to recompute the entire inverse matrix. We can use the Sherman-Morrison formula to update the old inverse Slater-matrix using the changes of the Slater-matrix. This turns out to be much less computationally expensive in most cases than recomputing from scratch the whole inverse Slater-matrix from the updated Slater-Matrix.

2.2 The Sherman-Morrison formula

Any change in a single row or column of a general $N \times N$ invertible matrix A can be expressed by a matrix U , constructed by the matrix product of two vectors u and v

$$U = uv^\top \quad (5)$$

where v^\top is the transpose of v . If there is a change in the m^{th} ($1 \leq m \leq N$) column of A , then v will be the m^{th} column of the $N \times N$ identity matrix, while u contains the changes to the matrix elements of A , such that the changed matrix \tilde{A}

$$\tilde{A} = A + U = A + uv^\top \quad (6)$$

A change like U is called a rank-1 update because the dimension of the image of U , when U is treated as an operator on a vector x , is 1

$$\dim(\text{img}(U)) = 1 \quad (7)$$

Another way of seeing this is to realise that all the rows/columns of U are linear combinations of each other, and therefore the dimension of the row/column space of U is 1.

The new inverse \tilde{A}^{-1} can then be computed from the old inverse A^{-1} with the following formula

$$\tilde{A}^{-1} = (A + uv^\top)^{-1} = A^{-1} - \frac{A^{-1}uv^\top A^{-1}}{1 + v^\top A^{-1}u} \quad (8)$$

This is in the literature called the Sherman-Morrison (SM) formula.

2.3 The iterative problem of SM in QMC=Chem

Multi-row or multi-column changes can be expressed by a rank- K matrix U , which is simply a sum of K rank-1 matrices U_k defined by

$$U = \sum_{k=1}^K U_k = \sum_{k=1}^K u_k v_k^\top \quad (9)$$

such that

$$\tilde{A} = \sum_{k=0}^K A_k = A_0 + \sum_{k=1}^K u_k v_k^\top \quad (10)$$

where A_0 is identified with the unmodified starting matrix A .

The SM-formula can be applied iteratively for each U_k until all K changes to A^{-1} have been applied

$$A_k^{-1} = A_{k-1}^{-1} - \frac{A_{k-1}^{-1} u_k v_k^\top A_{k-1}^{-1}}{1 + v_k^\top A_{k-1}^{-1} u_k}, \quad 1 \leq k \leq K \quad (11)$$

and A_K^{-1} is simply the final \tilde{A}^{-1} .

When K is small compared to the dimension N of the matrix (and N is not too small itself), computing the new inverse matrix using SM is much cheaper than inverting the whole matrix from scratch with routines like LAPACK that do a full LU-decomposition every time. However, the speed-up of applying a chain of K rank-1 updates with the SM comes with a cost.

The SM-method has a problematic property. When $K > 1$, a situation can arise where one of the U_k in the chain causes an A_k to become singular and $\det(A_k) = 0$. In that case A_k^{-1} is no longer defined, but when $\det(A_k)$ is close to zero, A_k^{-1} still exists, though it can have a large numerical error.

Even though the final inverse \tilde{A}^{-1} is guaranteed to exist, there is no such guarantee for the intermediaries A_k^{-1} . But there is a silver lining. Looking at Eqn. (10) we see that applying the updates to A is a commutative procedure; we may apply the update in whatever order we like.

Saying that $\det(A_k)$ zero is equivalent to saying that the denominator in Eq. (11) is zero. So we can introduce a ‘break-down’ parameter β such that

$$0 < \beta \ll 1 \quad (12)$$

where $\beta := 1 + v_k^\top A_{k-1}^{-1} u_k$. With this parameter we express what close to zero means. A value of $\beta = 1 \times 10^{-3}$ is customary in QMC=Chem. This value was also used during the experiments we did. Then for a given β , A_{k-1}^{-1} and u_k we stop updating when

$$\left| 1 + v_k^\top A_{k-1}^{-1} u_k \right| < \beta. \quad (13)$$

When this happens update u_k is put in a waiting queue. When all the updates are applied in the first pass, the waiting queue is processed in a second pass. The same can happen in the second pass and we end up with a new queue. Most of the time this process ends and eventually all updates are applied in some random order. But sometimes the queue cannot be emptied and thus a good order does not exist. When this happens a full inverse is computed from scratch. It might be that if we would have started with another u_k we could have found a good order. But this kind of algorithm would probably be much slower than simple computing a full inverse.

In the next section we will introduce several numerical methods that will either mitigate this problem or remove the possibility for it to happen completely.

3 Numerical methods

In this section we list and explain the numerical methods and algorithms that we developed to solve the numerical problems present in the current implementation of iterative SM in QMC=Chem. They are not necessarily listed in the order that we developed and tested them, but rather in order of ascending success rate.

To measure the success rate we used a straight forward method. We can define the following residual matrix

$$\rho := \tilde{A}^{-1}\tilde{A} - I \quad (14)$$

we can use ρ to define an error on \tilde{A}^{-1} . In the same way that we introduced the break-down parameter β before we define the following tolerance

$$0 < \tau \ll 1 \quad (15)$$

that will determine if the final \tilde{A}^{-1} is acceptable or not. Using the element-wise Max-norm we impose the following condition on the residual matrix ρ

$$\|\rho\|_{\max} < \tau \quad (16)$$

Whenever this inequality holds the numerical error on \tilde{A}^{-1} is acceptable. We have chosen value of $\tau = 1 \times 10^{-3}$, the same as the value of β . We choose this value because it was the typical order of magnitude of the norm of the residual matrix composed of the matrix/matrix-inverse pairs from the dataset extracted from QMC=Chem.

We now proceed to introduce the numerical methods we used and developed.

3.1 An naïve approach to iterative Sherman-Morrison

This method is the most simplistic of all the iterative methods and also the one that has the lowest success rate. It was inspired by Eq. (15) in P. Maponi's 2006 paper titled 'The solution of linear systems by using the Sherman-Morrison formula'[6]. This formula expresses the new inverse matrix A^{-1} as a product chain of intermediary updated inverse matrices $\{A_k^{-1}\}$

$$A^{-1} = \left(\mathbb{1}_N - \frac{A_{k-1}^{-1} u_k v_k^\top}{1 + v_k^\top A_{k-1}^{-1} u_k} \right) \cdots \left(\mathbb{1}_N - \frac{A_1^{-1} u_2 v_2^\top}{1 + v_2^\top A_1^{-1} u_2} \right) \left(\mathbb{1}_N - \frac{A_0^{-1} u_1 v_1^\top}{1 + v_1^\top A_0^{-1} u_1} \right) A_0^{-1} \quad (17)$$

It omits computing the solution and auxiliary solution vectors x_k and $y_{k,l}$ explained in Section 3.2. During the course of applying the chain of rank-1 updates, the denominators in Eqn. (17) are compared with β as in Eqn. (13). When the condition is true the update process is stopped.

For a given number of updates K , this kernel works as follows:

Algorithm 1: The “Naïve” kernel

Data: A_0^{-1} , $\dim A_0^{-1}$, $\det A_0^{-1}$, K , $\{u_k, v_k\}$, β
Result: A_K^{-1} , $\det A_K^{-1}$
initialisation: loop counter: $k \leftarrow 1$;
while $k \leq K$ **do**
 compute: $c_k \leftarrow A_{k-1}^{-1} u_k$ (column vector);
 compute the denominator: $d_k \leftarrow 1 + v_k^\top c_k$;
 if $|d_k| < \beta$ **then**
 exit;
 update: $\det A_{k-1}^{-1} \leftarrow \det A_{k-1}^{-1} \times d_k$;
 select the row from A_{k-1}^{-1} that is updated: $e_k \leftarrow v_k^\top A_{k-1}^{-1}$ (row vector);
 update: $A_{k-1}^{-1} \leftarrow A_{k-1}^{-1} - c_k d_k^{-1} e_k$;
 increment loop counter: $k \leftarrow k + 1$;

Even though it is the kernel with the lowest success rate we decided include it for the reason that if only one update is considered, it will not fail and is the fastest of all the listed kernels here, due to its low complexity.

3.2 Partial pivoting: Maponi’s method for solving linear systems

When we were first looking at the problem where a chosen order leads to a singular matrix in the course of applying a chain of rank-1 updates, we were guided by intuition to the notion of partial-pivoting. When performing Gaussian elimination on a matrix A to solve for the vector \mathbf{x} in the matrix equation

$$A\mathbf{x} = \mathbf{b} \quad (18)$$

the choice of pivot matters. Making a bad choice for the pivot element can lead to large errors in the solution due to propagation and amplification of round-off errors. In partial pivoting the row with the largest absolute value is chosen to minimise these errors.

The work of P. Maponi[6] is a logical starting point when considering partial pivoting in the context of using the Sherman-Morrison formula. Instead of computing the inverse directly he is using the Sherman-Morrison formula iteratively to solve a system of linear equations. He does this by defining the intermediate solution vectors x_k and the auxiliary solution vectors $y_{k,l} = A_k^{-1} u_l$. The inverse is then updated using the formula

$$A^{-1} = \left(\mathbb{1}_N - \frac{y_{M-1,p(M)} v_{p(M)}^\top}{1 + v_{p(M)}^\top y_{M-1,p(M)}} \right) \cdots \left(\mathbb{1}_N - \frac{y_{1,p(2)} v_{p(2)}^\top}{1 + v_{p(2)}^\top y_{1,p(2)}} \right) \left(\mathbb{1}_N - \frac{y_{0,p(1)} v_{p(1)}^\top}{1 + v_{p(1)}^\top y_{0,p(1)}} \right) A_0^{-1}. \quad (19)$$

We tested Algorithm 3 from his work, which used the idea of re-ordering the updates explained in Section 3.3. For a given update l , first all vectors $y_{k,l}$ are evaluated. The $y_{k,l}$ that gives the largest value for $|1 + v_l^\top A_k^{-1} u_l|$ is chosen and applied. This kernel is referred to as **Maponi A3**.

We show later that the numerical and computational performance of this method is not on par with the other ones presented in this report. Perhaps because of the added complexity to first obtain the solution \mathbf{x} via the auxiliary solutions $y_{k,l}$ and use them to reconstruct the intermediary $\{A_k^{-1}\}$ and A^{-1} .

For a given number of updates K , the kernel works as follows:

Algorithm 2: The “Maponi A3” kernel

Data: A_0^{-1} , $\dim A_0^{-1}$, K , $\{u_k, v_k\}$, β
Result: A_K^{-1}

for $k \leftarrow 1$ **to** K **do**

- | compute initial auxiliary sequence $\{y_{0,k}\}$: $y_{0,k} \leftarrow A_0^{-1}u_k$;
- | **for** $l \leftarrow 1$ **to** $K - 1$ **do**

 - | for a given l , select from $\{y_{l-1,k}\}$ the k that gives the largest value of:
 $d_k \leftarrow |1 + v_0^\top y_{l-1,k}|$;
 - | update: $A_{l-1}^{-1} \leftarrow A_{l-1}^{-1} - d_k^{-1} y_{l-1,k} v_k^\top A_{l-1}^{-1}$;
 - | **for** $k \leftarrow l + 1$ **to** K **do**

 - | compute next auxiliary sequence $\{y_{l,k}\}$:
 - | $y_{l,k} \leftarrow y_{l-1,k} - \frac{v_l^\top y_{l-1,k}}{1 + v_l^\top y_{l-1,l}} y_{l-1,l}$, $k \in \{l + 1, \dots, K\}$;

3.3 Reordering the updates

This is the approach currently used in QMC=Chem. It is an improvement of the “Naïve” method listed in Section 3.1. When an update u_k causes Eqn. (13) to be true, instead of exiting it is kept in a delay-queue. The next update is then evaluated and if it does not trigger a break-down, it is applied and the inverse is updated. If not, it is also send to the delay-queue. Then the next update is evaluated, etc.

When the whole list is evaluated the method considers the updates left in the delay-queue. If it is empty it is finished if it is not, the delay-queue is considered the new list of updates and they are re-evaluated. If one triggers a break-down it is again send to a delay-queue. The whole algorithm is repeated until the update queue is empty.

Sometimes this is not possible and the number of updates left in the queue is equal to the number that were going in. In this case the algorithm exits with an error. It is then upto the user what to do next. The algorithm is implemented as a recursive function in C.

For a given number of updates K , the kernel works as follows

Algorithm 3: The ‘‘Reordering’’ kernel

Data: A_0^{-1} , $\dim A_0^{-1}$, $\det A_0^{-1}$, K , $\{u_k, v_k\}$, β
Result: A_K^{-1} , $\det A_K^{-1}$

initialisation:
loop counter: $k \leftarrow 1$;
number of later updates: $L \leftarrow 0$;
later updates queue: $\{u_l, v_l\} \leftarrow \emptyset$;

while $k \leq K$ **do**

compute: $c_k \leftarrow A_{k-1}^{-1} u_k$ (column vector);
compute the denominator: $d_k \leftarrow 1 + v_k^\top c_k$;
if $|d_k| < \beta$ **then**

add update u_k to the queue: $\{u_l, v_l\} \leftarrow u_k, v_k$;
increment number of later updates: $L \leftarrow L + 1$;
increment loop counter: $k \leftarrow k + 1$;
continue to the next iteration;

update: $\det A_{k-1}^{-1} \leftarrow \det A_{k-1}^{-1} \times d_k$;
select the row from A_{k-1}^{-1} that is updated: $e_k \leftarrow v_k^\top A_{k-1}^{-1}$ (row vector);
update: $A_{k-1}^{-1} \leftarrow A_{k-1}^{-1} - c_k d_k^{-1} e_k$;
increment loop counter: $k \leftarrow k + 1$;

if $L = K$ **then**

exit;

else if $L > 0$ **then**

recursive call using A_{K-L}^{-1} , $\dim A_{K-L}^{-1}$, $\det A_{K-L}^{-1}$, L , $\{u_l, v_l\}$;

3.4 Slagel’s method of update splitting

This method is based on the ‘reordering’ approach of Section 3.3. However, it is improved with an implementation of Joseph Tanner Slagel’s splitting method[8].

The improvement is surprisingly simple. Once an update u_k causes Eqn. (13) to be true, instead of only pushing the update to the delay-queue, the update is first split in half, $u_k \rightarrow \frac{1}{2}u_k$. Then the first half of u_k is applied since it can no longer cause Eqn. (13) to be true. Then the other half is push on the delay-queue. Then the next update is considered. Either it is applied entirely, or split in half. The algorithm is then executed recursively until the delay-queue is empty.

It can be formally proved that by using this method it will take a finite number of splits to apply any chain of rank-1 updates, as long as the fully updated matrix is guaranteed to have an inverse. If you are interested in reading the proof, you can have a look at Theorem 3.1.4 in Section 3.1.2 of [8].

For a given number of updates K , the kernel works as follows

Algorithm 4: The “Splitting” kernel

Data: A_0^{-1} , $\dim A_0^{-1}$, $\det A_0^{-1}$, K , $\{u_k, v_k\}$, β

Result: A_K^{-1} , $\det A_K^{-1}$

initialisation:

loop counter: $k \leftarrow 1$;

number of later updates: $L \leftarrow 0$;

later updates queue: $\{u_l, v_l\} \leftarrow \emptyset$;

while $k \leq K$ **do**

compute: $c_k \leftarrow A_{k-1}^{-1} u_k$ (column vector);

compute the denominator: $d_k \leftarrow 1 + v_k^\top c_k$;

if $|d_k| < \beta$ **then**

split the current update in half: $c_k \leftarrow \frac{1}{2}c_k$;

add half of update u_k to the queue: $\{u_l, v_l\} \leftarrow \frac{1}{2}u_k, v_k$;

update the denominator: $d_k \leftarrow 1 + v_k^\top c_k$;

increment number of later updates: $L \leftarrow L + 1$;

update: $\det A_{k-1}^{-1} \leftarrow \det A_{k-1}^{-1} \times d_k$;

select the row from A_{k-1}^{-1} that is updated: $e_k \leftarrow v_k^\top A_{k-1}^{-1}$ (row vector);

update: $A_{k-1}^{-1} \leftarrow A_{k-1}^{-1} - c_k d_k^{-1} e_k$;

increment loop counter: $k \leftarrow k + 1$;

if $L > 0$ **then**

recursive call using A_{K-L}^{-1} , $\dim A_{K-L}^{-1}$, $\det A_{K-L}^{-1}$, L , $\{u_l, v_l\}$;

3.5 All at once: the Woodbury matrix identity

To circumvent the problem of iterative SM altogether we can use its generalised form, called the Woodbury (WB) Matrix Identity. The SM-formula is simply the special case for a rank-1 update matrix. This is its most general form

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \quad (20)$$

where $U : N \times K$, $C : K \times K$ and $V : K \times N$ are conformable matrices. The matrices U and V contain the updates and the update locations in the Slater-matrix respectively. Notice that

$$D := C^{-1} + VA^{-1}U \quad (21)$$

is at most $K \times K$. To give some perspective, when calculating the electronic structure and ground state properties of the compound Benzene with QMC=Chem, in 75% of the cases the number of updates that need to be applied to the inverse Slater-matrix is not larger than $K = 6$, whereas the Slater-matrix is $N = 21$.

Having said that, using Woodbury for rank- K updates can be more costly than K rank-1 SM-iterations, depending on the architecture and the implementation. There is no way to know a priori if this will be the case, so we recommend running a benchmark for an SM-kernel and a Woodbury-kernel and make the final choice based on the results.

Notice also that for $C = I$, $U = u$ and $V = v^\top$, D reduces to

$$D = 1 + v^\top A^{-1}u \quad (22)$$

and since $v^\top A^{-1}u$ is scalar, we recover the SM-denominator $D^{-1} = 1 / (1 + v^\top A^{-1}u)$.

In our implementations we use $C = I$, so the WB-formula we use to compute the new inverse is

$$A_{\text{new}}^{-1} = A_{\text{old}}^{-1} - BD^{-1}E \quad (23)$$

where

$$B := A_{\text{old}}^{-1}U \quad (24)$$

$$D := I + VB \quad (25)$$

$$E := VA_{\text{old}}^{-1} \quad (26)$$

The algorithm for K updates is listed below

Algorithm 5: The “ $K \times K$ Woodbury” kernel

Data: A_{old}^{-1} , $\dim(A_{\text{old}})$, $\det(A_{\text{old}})$, K , U, V, β
Result: A_{new}^{-1} , $\det(A_{\text{new}})$
compute: $B \leftarrow A_{\text{old}}^{-1}U$;
compute: $D \leftarrow I + VB$;
compute: $E \leftarrow VA_{\text{old}}^{-1}$;
compute: $\det(D)$ from LU factorization;
if $|\det(D)| < \beta$ **then**
 └ exit;
update: $\det(A_{\text{new}}) \leftarrow \det(A_{\text{old}}) \times \det(D)$;
compute: D^{-1} from earlier LU factorization;
compute: $A_{\text{new}}^{-1} \leftarrow A_{\text{old}}^{-1} - BD^{-1}E$;

4 HPC versions

To run with optimum efficiency on HPC architectures we used several well known techniques. Most of the time was spent optimising the vectorisation that is discussed next. Manual loop-unrolling for a fixed number of updates in the WB-kernel is discussed in Section 4.2. To fine tune the performance even more and have specialised kernels for every system size we use source-code generation that is discussed in Section 4.3.

4.1 Vectorisation

To vectorise the loops fully a few things need to be setup correctly simultaneously. Only then all the obstacles for the compiler are removed and it can do its work the way we want.

4.1.1 Zero-padding of arrays

Loops in our SM kernel were not automatically vectorised completely when the leading dimension of an array was not an integer multiple of the CPU’s vector register size. In this case a tail-loop was inserted and this has a negative impact on performance. It is then better to add a small amount of extra work by padding the arrays with extra zeros, so that their leading dimension becomes an integer multiple of the vector register size. Then tail-loops are unnecessary. Even though more work is done, vector-instructions are usually faster than scalar-instructions.

The amount of zero-padding depends on the size of vector registers of the CPU. For example for an Intel CPU with AVX2 extensions, the size of a vector register is 256 bits. This is equivalent to 8 ints/floats or 4 long ints/doubles 64 bit machine. For AVX512 these numbers double.

The datasets we used to benchmark the performance of our kernels were all zero-padded and in the argument list of the kernel calls there are always two arguments that need to be passed. The real dimension of the arrays and the zero-padded dimension, called the leading dimension. A formal treatment of zero-padding can be found in Appendix A.

4.1.2 Pointer aliasing

The compiler is only able to vectorise the loops properly if it knows that the pointers that point to the arrays are pointing to distinct blocks of memory. In other words, the pointers cannot at any single point during the execution of the function body point to the same memory address.

If the compiler is not sure of this and the memory of one of the arrays could be accidentally overwritten by using the pointers to other arrays, it will insert extra assembly instructions to make sure that the intended result is guaranteed. This will of course come with a performance cost so we need to prevent this from happening.

Luckily there is a way to tell the compiler that pointers do not alias by using the `__restrict` keyword in the pointer arguments of the function header. For example, for two `double`-pointers this could look like:

```
void some_function(double* __restrict DATA, double* __restrict OTHER_DATA);
```

4.1.3 Contiguous memory and memory alignment

A final issue that can arise when vectorising loops is when the memory of the array is not contiguous and/or the data in the array is not aligned to memory addresses that are predictable. In this case simple pointer arithmetic is not possible and the compiler needs to insert extra assembly code to make sure it fetches the correct data. In that case vector instructions cannot be used and therefore the final code will only be partially vectorised, or not at all.

Again, we helped the compiler by making sure that the arrays contiguous in memory and that the data addresses is properly aligned to n-byte boundaries by declaring e.g., for a K element `double`-array `Array`:

```
double __attribute__(aligned(8)) Array[K];
```

4.2 Manual loop-unrolling: 2×2 and 3×3 WB and the Blocking kernel

For the Woodbury kernel we have improved the performance by considering special cases. Looking at Fig. 5.1, it is evident that most of the time the number of updates that have to be processed is small, mostly one, two or three updates. It is for that reason that we made two special cases of the $K \times K$ Woodbury kernel: a 2×2 case and a 3×3 case. Because the number of rank-1 updates are known we can unroll and simplify some loops. It also permits us to use explicit expressions for computing the determinants and inverses of the 2×2 and 3×3 matrix D of Eq. (21), removing the need for expensive calls to LAPACK for the LU-decomposition and matrix inversion.

To deal with all the possible cases we made a kernel that combines the Splitting kernel with the WB 2×2 and 3×3 kernels that we call the “Blocking”-kernel.

To maximise arithmetic intensity the total number of updates is divided first in blocks of 3 rank-1 updates. Each of these blocks are then send to the WB 3×3 -kernel. If any of these

blocks fail due to $|\det D| < \beta$, the updates in the block are attempted with the Splitting-kernel and the split updates are moved to a queue for later.

After all blocks are done we check if the remainder is either 2 or 1 rank-1 updates. If the remainder is 2, the updates are send to the WB 2×2 -kernel. If this kernel fails due to $|\det D| < \beta$, the last 2 rank-1 updates are also attempted with the Splitting-kernel, any split updates are also moved to the queue for later.

If the remainder is 1, the last update attempted with Splitting-kernel, again adding half of it to the queue in case of a split.

Then finally, if there are any, the remaining halves in the queue are sent to the Splitting kernel which should always converge.

There is also a special case for 4 rank-1 updates. Performance measurements have shown that doing 4 rank-1 updates in 2 blocks of 2 rank-1 updates is faster than doing them in a block of 3 and the remaining one with the Splitting kernel. The method is the same as for the general case: if a block of 2 rank-1 updates fails, it is sent to the Splitting kernel and any split updates are moved to a queue to processed in the end.

The algorithm looks as follows:

Algorithm 6: The “Blocking” kernel

Data: A^{-1} , $\dim A^{-1}$, $\det A^{-1}$, K , U, V, β
Result: \tilde{A} , $\det \tilde{A}$

first initialisation:

number of later updates: $L \leftarrow 0$;
 later updates queue: $\{u_l, v_l\} \leftarrow \emptyset$;

if $K = 4$ **then**

- call** WB 2×2 with the **first** 2 rank-1 updates;
- if** WB 2×2 *fails* **then**
 - call** Splitting-kernel¹ with the **first** 2 rank-1 updates;
 - if** *any splits* **then**
 - add half of those updates u_k to the queue: $\{u_l, v_l\} \leftarrow \frac{1}{2}u_k, v_k$;
 - increment number of later updates: $L \leftarrow L + (1 \vee 2)$;
- call** WB 2×2 with the **last** 2 rank-1 updates;
- if** WB 2×2 *fails* **then**
 - call** Splitting-kernel¹ with the **last** 2 rank-1 updates;
 - if** *any splits* **then**
 - add half of those updates u_k to the queue: $\{u_l, v_l\} \leftarrow \frac{1}{2}u_k, v_k$;
 - increment number of later updates: $L \leftarrow L + (1 \vee 2)$;
- if** $L > 0$ **then**
 - call** Splitting-kernel² using A_{K-L}^{-1} , $\dim A_{K-L}^{-1}$, $\det A_{K-L}^{-1}$, L , $\{u_l, v_l\}$;
- exit**;

second initialisation:

number of 3 rank-1 update blocks: $N \leftarrow \lfloor K/3 \rfloor$;
 remainder: $R \leftarrow K \bmod 3$;

for *number of blocks N* **do**

- call** WB 3×3 for each successive block of 3 rank-1 updates;
- if** *WB* 3×3 *fails* **then**
 - call** Splitting-kernel¹ with the 3 rank-1 updates of the current block;
 - if** *any splits* **then**
 - add half of those updates u_k to the queue: $\{u_l, v_l\} \leftarrow \frac{1}{2}u_k, v_k$;
 - increment number of later updates: $L \leftarrow L + (1 \vee 2 \vee 3)$;

if $R = 2$ **then**

- call** WB 2×2 with the **last** 2 rank-1 updates;
- if** *WB* 2×2 *fails* **then**
 - call** Splitting-kernel¹ with the **last** 2 rank-1 updates;
 - if** *any splits* **then**
 - add half of those updates u_k to the queue: $\{u_l, v_l\} \leftarrow \frac{1}{2}u_k, v_k$;
 - increment number of later updates: $L \leftarrow L + (1 \vee 2)$;

else

- call** Splitting-kernel¹ with the **last** rank-1 update;
- if** *a split* **then**
 - add half of the update u_k to the queue: $\{u_l, v_l\} \leftarrow \frac{1}{2}u_k, v_k$;
 - increment number of later updates: $L \leftarrow L + 1$;

if $L > 0$ **then**

- call** Splitting-kernel² using A_{K-L}^{-1} , $\dim A_{K-L}^{-1}$, $\det A_{K-L}^{-1}$, L , $\{u_l, v_l\}$;

exit;

Actually, only the non-recursive part of the Splitting kernel is used here. It doesn't call itself after it applied and/or split the updates. It only populates the update queue and increment the number of later updates if there are splits.

Here the full recursive version of the Splitting kernel is used.

4.3 Explicit array dimensions and kernel-template generation

When the array dimensions and loop bounds are known to the compiler at compile time it can optimise more aggressively for loop-unrolling and vectorisation, greatly reduce the number of assembly instructions and increase the overall performance.

To that end a Python script is used to automatically generate C-source code for each specific system size. This allows the C-compiler to know always the array dimensions and loop-bounds at compile time. For the users a general kernel will be exposed in the QMCkl that contains a switch-mechanism that calls the internal specialised kernels that were generated by the Python script. A simple example of how this is done in C, for the fictitious kernel `kernel(size, other_parameters)`, is shown in Listing 1.

As of the writing of this report, the automatic generation and selection of size-specific kernels is not able to deal yet with array padding. This is still a work in progress and will be added to the QMCkl when it is ready.

```

double kernel(size, other_parameters)
{
    switch(size)
    {
        case 1: return kernel_1(other_parameters);
        case 2: return kernel_2(other_parameters);

        ...
        ...
        ...

        case n: return kernel_n(other_parameters);
    }
}

```

Listing 1: Automatic kernel selection based on the function argument `size`.

5 Experiments with QMC=Chem

QMC=Chem relies heavily on the use of SM for updating the determinants and inverses in the many-body wave function. So any gain in numerical accuracy and/or performance could have a significant impact on its overall performance.

Because we want to know how our algorithms would behave in QMC=Chem we tested them using real data extracted from a short run of QMC=Chem on the Benzene molecule. This way we can also account for the behaviour of the kernels due to any particularities of the data itself. How we extracted the dataset will be explained in the next section.

We then proceed with explaining the exiting problems with the current implementation of the SM-method in Section 5.2. After that we elaborate on the measurement procedure and give details about the machine architecture on which the measurements have been performed in Section 5.3. Finally the results will be presented in Sections 5.4 and 5.5.

5.1 Dataset extraction

We tested our kernels on datasets[5] that we extracted from QMC=Chem using Variational Monte Carlo (VMC) to run a ground state calculation for Benzene (42 electrons, 21 spin-up, 21 spin-down), using 329 and 15784 determinants respectively. The datasets are extracted after a single electron move, when the MOs and determinants in Eqn. 1 are being updated. Additional Fortran `write`-statements were added to the file `$QMCCHEM_ROOTsrc/det.irp.f` to extract and save the following data

- the number of the determinant in the chain, starting from 1 and omitting numbers $i \times N_{\text{dets}}$, $i = 0, 1, \dots$ that are computed with LAPACK
- the dimension of the Slater-matrix
- the number of rank-1 updates (= number of MO/column changes)
- the Slater-matrix S before update
- the inverse Slater-matrix before update

- the column-indices of the updates
- the the updates themselves as replacement-updates

Fig. 5.1 shows the distribution of the occurrence frequency of the number of rank-1 updates. Both the datasets have a maximum of 15 rank-1 updates per changed determinant. The only difference is their relative occurrence frequency, for the 329-determinant dataset 1–6 rank-1 updates account for 75% of the cases, whereas for the 15784-determinant dataset it is 1–8 rank-1 updates.

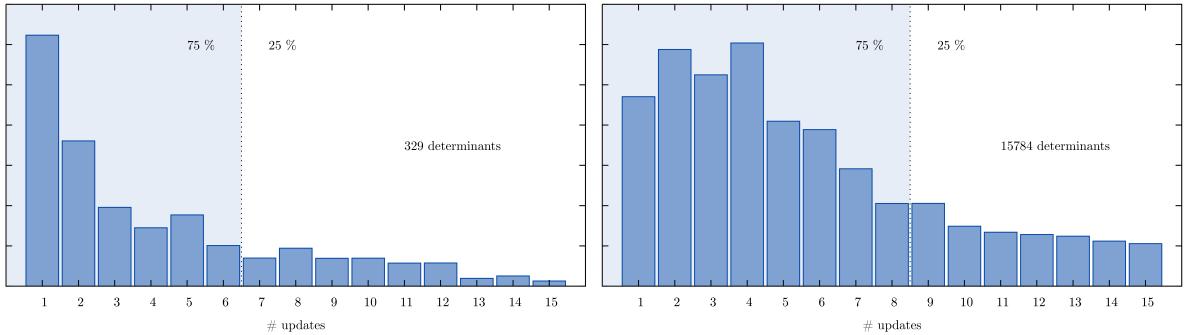


Figure 5.1: Distribution of the occurrence frequency for each number of rank-1 updates that can occur. The blue area represents the 3rd quartile of the distribution. For the 329-determinants case, 75% of the time updates consist of 1–6 rank-1 updates. For the 15784-determinants case updates consist of 1–8 rank-1 updates.

5.2 Problems in the current implementation

We now continue to list the issues in the current implementation of iterative Sherman-Morrison in QMC=Chem. All of these issues have been addressed in our improved kernels.

5.2.1 Issue 1: Numerical accuracy with large number of determinants

The SM is implemented in QMC=Chem at this moment is equivalent to the reordering method described in Section 3.3. In case there is one determinant along the chain that has more than one rank-1 update and a suitable order to apply the U_k cannot be found the whole inverse is recomputed with LAPACK. Then the next determinant in the chain is again updated using SM, if possible. If not, LAPACK, etc. In our particular case for Benzene with 329 determinants and the used value of $\beta = 1 \times 10^{-3}$ in QMC=Chem, LAPACK was never called mid-chain. But for the case of 15784 determinants, numerical accuracy suffers too much before the end of the chain is reached. LAPACK is called about 20 times per 15784-determinant chain.

Because of this, QMC=Chem is loosing more time during LAPACK calls then if it were to use only Sherman-Morrison. The challenge is therefore to first increase the numerical accuracy of the SM method. Once that is achieved, calls to LAPACK should only occur for the first determinant in the chain and thereby increasing the overall performance.

If we can improve the numerical accuracy of the kernels, calls to LAPACK mid-chain could be reduced as well as is part of its performance cost.

5.2.2 Issue 2: Low arithmetic intensity (HPC)

All of the SM methods discussed below have one property in common in that they consist mostly of Matrix-Vector operations (BLAS Level 2). To increase the performance of applying multiple

rank-1 updates even more it would be fruitful if we could somehow increase the arithmetic intensity by using Matrix-Matrix operations (BLAS Level 3). One way of doing that is to use the more general Woodbury matrix identity discussed in Section 3.5 or 4.2.

5.3 Measurement machine architecture and measurement procedure

All the measurements have been performed on Intel-based PC with a Core i9-10900 64-bit CPU and 32 GB of non-ECC memory. The CPU has the following vector instruction extensions: SSE4.1, SSE4.2 and AVX2.

In the experiments with Benzine, the Slater-matrices will receive at most 15 rank-1 updates. One series of $1 \leq k \leq 15$ rank-1 is called an *update cycle*. For each update cycle we read the following data from the dataset (except D):

D : dimension of the Slater-matrix (only once at the beginning)

S : the current Slater-matrix

S^{-1} : the current inverse Slater-matrix

K : the number of rank-1 updates

C : the vector of length K containing the column numbers that need to be updated

U : $K \times D$ matrix containing the column updates

During the measurement, the following data is written to `STDOUT`

`CYCLE` cycle number

`UPDS` number of updates K in the cycle

`ERR_IN` check on the *input* matrices S and S^{-1} : 0 if $\|\rho_{\text{in}}\| < \tau$, 1 otherwise

`ERR_BREAK` check on the SM or WB denominator during update: 0 if $|1 + v_k^\top A_{k-1}^{-1} u_k| > \beta$, 1 otherwise

`ERR_OUT` check on the *output* matrices S and S^{-1} : 0 if $\|\rho_{\text{out}}\| < \tau$, 1 otherwise

`SPLITS` number of splits for kernels that use Slagel-splitting. *This is a global variable that is set to zero at the start and updated during kernel execution.*

`BLK_FAILS` number of failed blocks of 3- or 2 rank-1 updates for the Blocking kernel. *This is a global variable that is set to zero at the start and updated during kernel execution.*

`MAX` element-wise max-norm of ρ_{out}

`FROB` Frobenius-norm of ρ_{out}

`COND` Condition number $\|S^{-1}S\|$ of the output matrices

`CPU_CYC` Number of CPU cycles for the whole update cycle

`CPU_CYC/UPD` Number of CPU cycles / K

`CUMUL` Cumulative of all update cycles

`REC` Number of recursions for recursive kernels. *This is a global variable that is set to zero at the start and updated during kernel execution.*

Then the measurement program will do the following, based on *the chosen kernel* and N that has to be set at execution:

```

Set the global cumulator to 0;
for each update cycle do
    1. Read  $S$ ,  $S^{-1}$ ,  $K$ ,  $C$  and  $U$  from dataset
    2. Check  $\|\rho_{\text{in}}\| < \tau$  and write to ERR_IN
    3. Set CPU_CYC and SPLITS to 0

for  $N$  repetitions do
    Make a copy  $S_{\text{copy}}^{-1}$  of  $S^{-1}$  and use it in the chosen kernel;
    for the chosen kernel do
        1. Poll the CPU cycle counter
        2. Execute the chosen kernel and record the exit status in ERR_BREAK
        3. Poll the CPU cycle counter again
        4. Add CPU cycle difference to accumulator

    1. Copy the updated  $S_{\text{copy}}^{-1}$  back to  $S^{-1}$ 
    2. Divide cycle- and split-accumulator by  $N$  and record in CPU_CYC and SPLITS
    3. Add CPU_CYC to the global cumulator and record in CUMUL
    4. Divide cycle-accumulator by  $K$  and record in CPU_CYC/UPD
    5. Update  $S$ 
    6. Check  $\|\rho_{\text{out}}\| < \tau$  and write to ERR_OUT
    7. Write the recorded measurements to STDOUT

```

5.4 Results for the numerical accuracy

To measure the numerical accuracy we use the two parameters β and τ defined in Eqns. (12, 15). It has been determined empirically by use of QMC-Chem that a good “near zero” value, below at which the quality of the updated inverse suffers too much, is $\beta = 1 \times 10^{-3}$. After a kernel has updated the inverse Slater-matrix the residual ρ is computed and the inequality Eqn. (16) is evaluated with $\tau = 1 \times 10^{-3}$. When the inequality holds the inverse Slater-matrix is considered numerically acceptable and **PASSES**. When the inequality doesn’t hold, the inverse is considered numerically unacceptable and **FAILS**.

The number of **PASSES** and **FAILS** are collected and with them we define the following fail rate

$$\text{Fail rate} = \frac{\# \text{ of FAILS}}{\# \text{ of PASSES} + \# \text{ of FAILS}} \times 100\% \quad (27)$$

5.4.1 329 α -determinants

In Table 5.1 the fail rates are shown for 10384 update cycles and 329 determinants in the wave function. The Naïve-kernel is particularly bad because as the number of rank-1 updates increases along the 329-determinant chain the probability of encountering a rank-1 update that cause singular behaviour increases as well. The Reordering-kernel is already two orders of magnitude better, having only 26 failed updates. The Splitting- and Blocking-kernel are clearly the best, with only 21 failures. Unlike the failures for the Naïve-kernel, these are not failures of the kernels themselves due to denominators that are too small, i.e. condition ??, but due to a too large max norm of the residual matrix ρ .

Kernel	# Pass	# Fail	Failrate (%)
Naïve	6598	3786	36.46
Reordering	10358	26	0.25
Splitting	10363	21	0.20
Blocking & Splitting	10363	21	0.20
Maponi A3	9544	840	8.09
Maponi A3 & Splitting	9921	463	4.46

Table 5.1: Fail rates for Benzene with 329 α -determinants and 10384 update cycles in total, $\beta = 1 \times 10^{-3}$ and $\tau = 1 \times 10^{-3}$.

Numerical drift For the case of 329-determinants there is a general trend that the max norm of the residual matrix $\|\rho\|_{\max}$ steadily increases while going along the chain, and then falls back again after the next first determinant is computed with LAPACK. This is shown in Fig. 5.2. This suggest the numerical noise is slowly accumulated and carried along the chain. Unfortunately the failing cases are not in general situated near the ends of chains so it might be that they were simply caused by particularly ill conditioned input matrices.

Numerical accuracy independent of number of updates Another interesting aspect of the SM and WB kernels is that the numerical accuracy, measured by the Max-norm of the residual matrix ρ does not seem to dependent on the number of rank-1 updates they have to apply. This is showed in Figure 5.3. For most kernels and number of updates the Max-norm appears the be centred around 1×10^{-5} . There is one extra branch at 1 rank-1 update centred around 1×10^{-11} . These correspond to the second determinants in the chain as discussed just above.

5.4.2 15784 α -determinants

For this case we chose to show a slightly larger level of detail by giving the fail rates for five representative number of rank-1 updates: an overall fail rate all , the case where there is only 1 rank-1 update to compare against the Naïve kernel, 2 to compare against Woodbury 2×2 (WB2), 3 to compare against Woodbury 3×3 (WB3) and 6 to see how well the Blocking kernel performs against Reordering/Splitting. The results are summarised in Table 5.2. As in the 329-determinant case, the Naïve kernel is by far the worst as rank- k , with $k > 1$, are very common. It is surprising however that for the rank-1 update case the Naïve kernel seems to do ever so slightly worse than the Splitting kernel, but maybe we are actually looking at the machine noise level; they are the same upto 4 decimal digits. For the rank-2 update case WB2 is slightly worse than Splitting, Reordering and Blocking because the Woodbury kernels seem

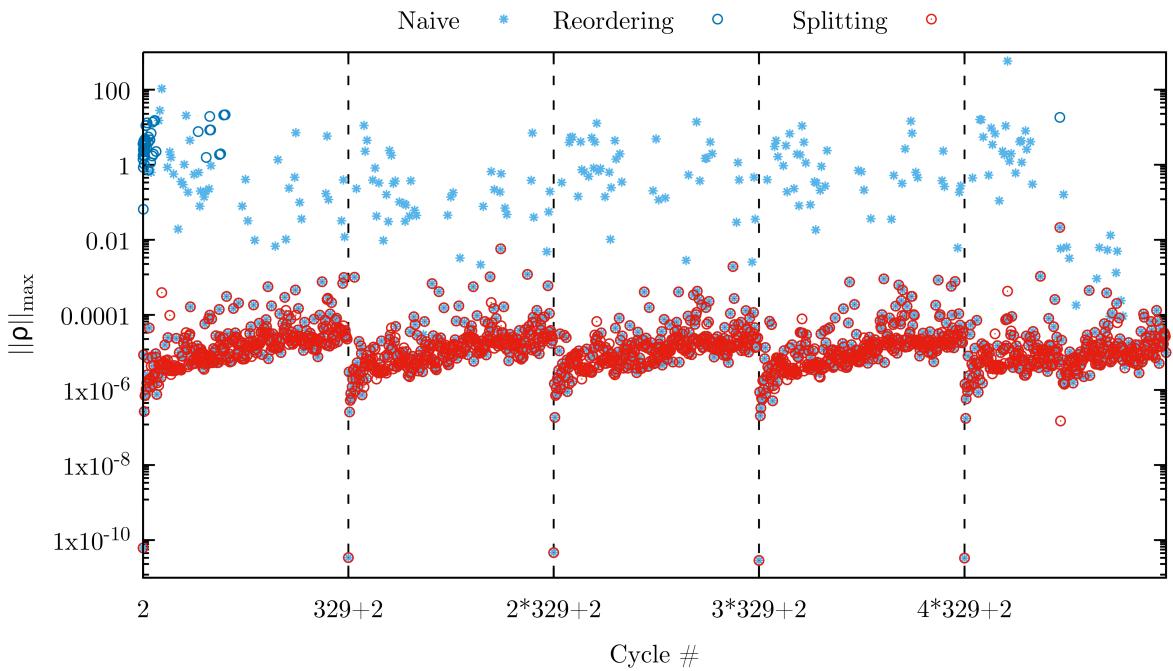


Figure 5.2: Numerical drift, steadily increasing from the beginning to the end of the 329 determinant update chain.

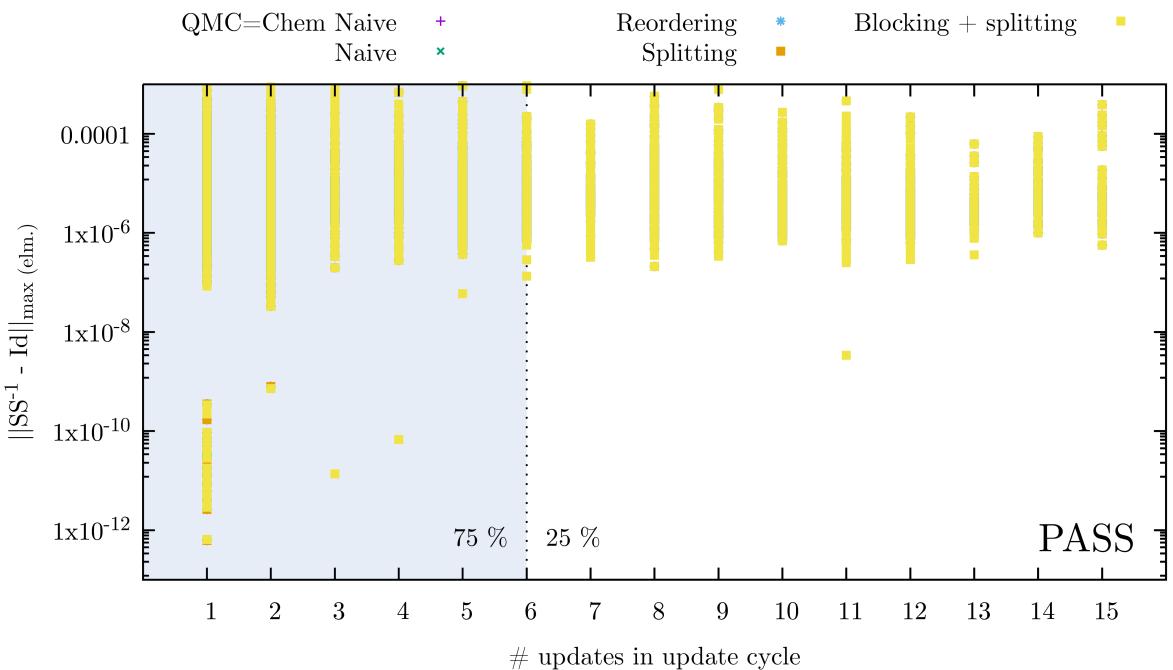


Figure 5.3: The influence of the number of rank-1 updates to numerical accuracy for various kernels.

cause slightly larger loss of precision than the SM-only kernels. The Blocking kernel behaves the same as the Splitting-kernel because it retries the failed cases of WB2 (because the denominator is too small) by sending them to the Splitting kernel and then passes. For WB3 we see the same behaviour for the rank-3 update cases. The failing blocks are retried using the Splitting kernel. Overall the Blocking and Splitting kernels behave numerically equivalent and are the best compared to the others. The Blocking kernel is a few more failing cases because WB2 and WB3 are slightly less accurate, but as we will see later this comes with a slight increase in speed.

# upds (222008 cycles)	Kernel:	Naïve	Splitting	Reordering	WB2	WB3	Blocking
all (222008 cycles)		48.780	.831	.932	—	—	.831
1 (23533 cycles)		.939	.931	.939	—	—	.931
2 (29388 cycles)		5.962	.759	.769	.783	—	.759
3 (26239 cycles)		54.545	.808	.873	—	.842	.808
6 (19442 cycles)		73.835	.741	.905	—	—	.746

Table 5.2: Fail rates for Benzene with 15784 α -determinants, 222008 determinants in total, $\beta = 1 \times 10^{-3}$ and $\tau = 1 \times 10^{-3}$. The fail rates are shown for five relevant number of updates.

5.5 Results for the performance

Before we jump to the final performance results there are two aspects that are interesting in their own respect. One of them is the performance per rank-1 update. Section 5.5.1 will show that this is mostly a constant. Which is good news. Another aspect, related to the first, is performance stratification. This is explained in Section 5.5.2.

5.5.1 Performance per rank-1 update

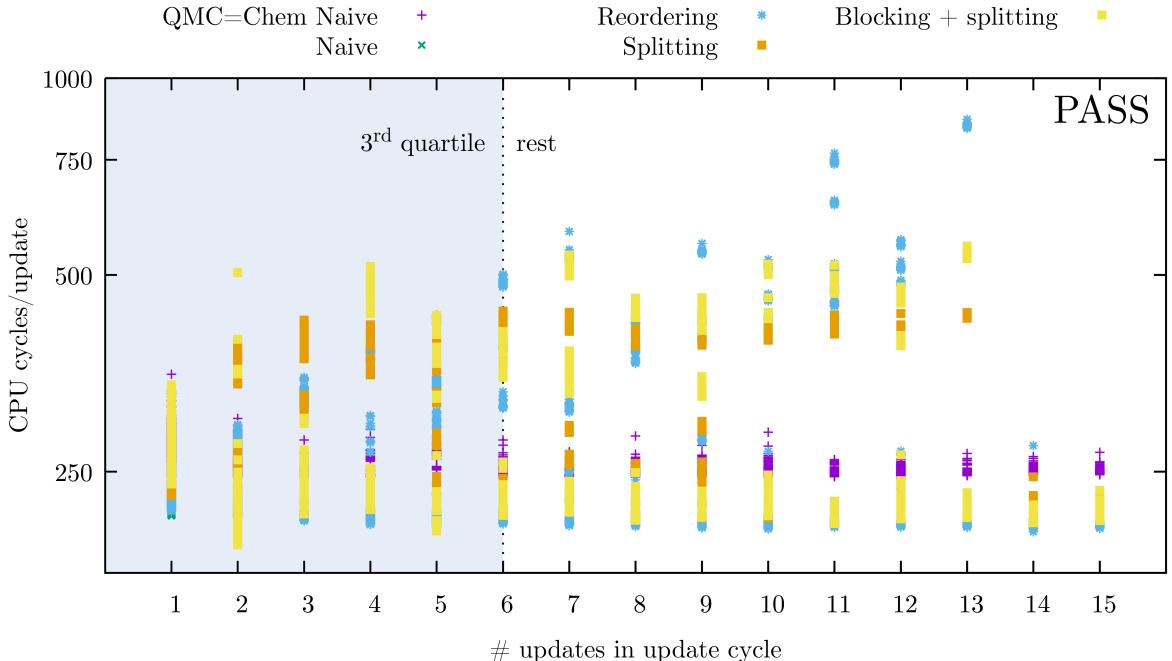


Figure 5.4: Performance in CPU-cycles per rank-1 update for various kernels.

In Fig. 5.4 is shown the number of CPU-cycles per rank-1 update versus the number of rank-1

updates for various kernels. Focussing our attention on the 3rd there are two main branches. One is centred around 250 CPU-cycles/rank-1 update. For one rank-1 update the spread is due to variability of the machine itself. For the cases of more than one rank-1 update this is mainly due to the number of splits for the Splitting kernel, or the number of failed blocks in the Blocking kernel.

Looking at the left two panels of Fig. 5.5, for the case of Benzene, 329 determinants the maximum number of splits is always one less than the number of updates, in case of the Splitting kernel. The Blocking kernel seems to get away with less splits in general, except in the case of two rank-1 updates. On the right two panels of Fig. 5.5 we plotted the average number of CPU cycles per rank-1 update as a function of the number of splits. In the PASSING cases the number of CPU cycles per update increase linearly as the number of splits increases, up until 6 splits where it plateaus. For the FAILING cases there are never more than 3 splits, but most of these occurrences are clustered around 0–2 splits and take an average of about 300 CPU cycles per update.

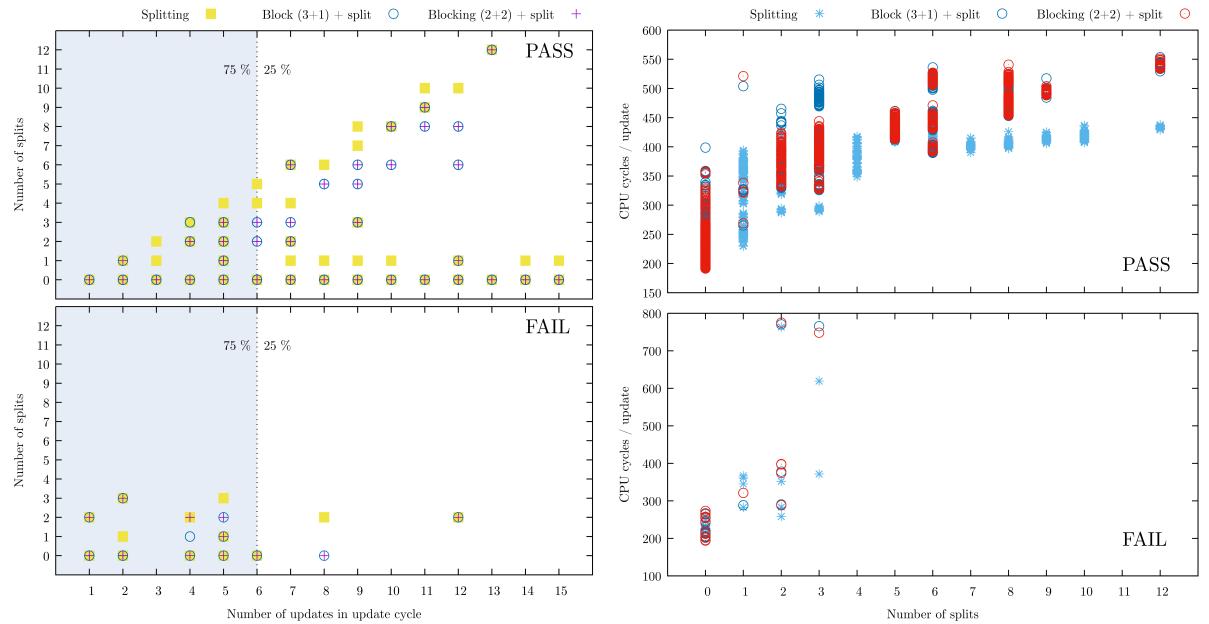


Figure 5.5: Distribution of the occurrence frequency for each number of rank-1 updates that can occur. The blue area represents the 3rd quartile of the distribution. For the 329-determinants case, 75% of the time updates consist of 1–6 rank-1 updates. For the 15784-determinants case updates consist of 1–8 rank-1 updates.

5.5.2 Performance stratification

When we plot the element-wise max-norm of the residual matrix ρ against the number of CPU cycles for all kernels, we get the picture that is shown in Figure 5.6. Each coloured point on this plane corresponds to exactly one update chain computed for the kernel corresponding to that colour. Where this point is located on the plain, together with all the others of the same colour, eventually determines the numerical accuracy and performance of the kernel.

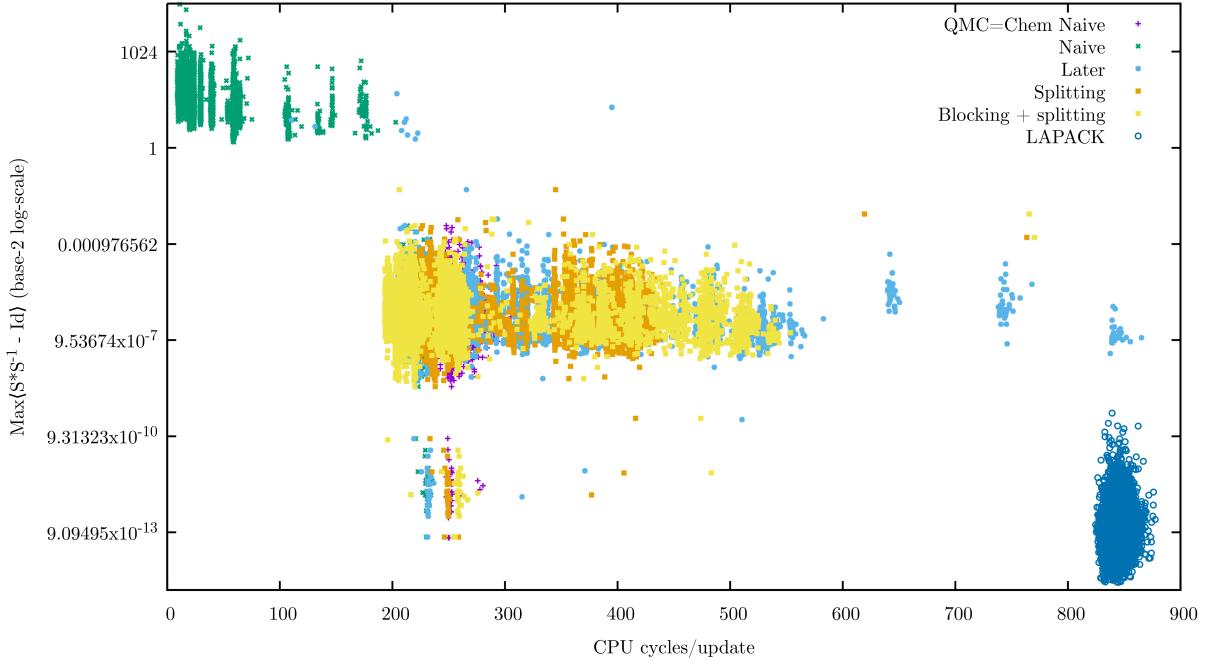


Figure 5.6: The element-wise max-norm of the residual matrix ρ as a function of the number of CPU cycles (averaged) for all tested kernels.

When looking at Figure 5.6 carefully we can see clusters of vertical stripes, except for the LAPACK kernel. Each of these clusters corresponds to a specific set of circumstances. E.g. the green cluster of stripes in the top-left corner are all above the tolerance threshold τ and are failing update-chains. The vertical stripes are formed by how many updates were applied successfully by the “Naïve kernel” before a break-down occurred.

The cluster in the bottom-left between 200 and 300 CPU cycles/update are the update chains that correspond to the first ones after a full inversion with LAPACK and have only one rank-1 update. So they did not have the time to accumulate any numerical noise, hence their relatively small residual.

Then there is the big cluster in the middle covering most remaining cases. The light blue stripes are formed by the recursions of the “Reordering” kernel. Some update chains are harder than others so sometimes it takes more than a few recursive calls to find a successful order. This accounts for the rather wider range of required CPU cycles/update than that of the “Splitting” and “Blocking+splitting” kernels.

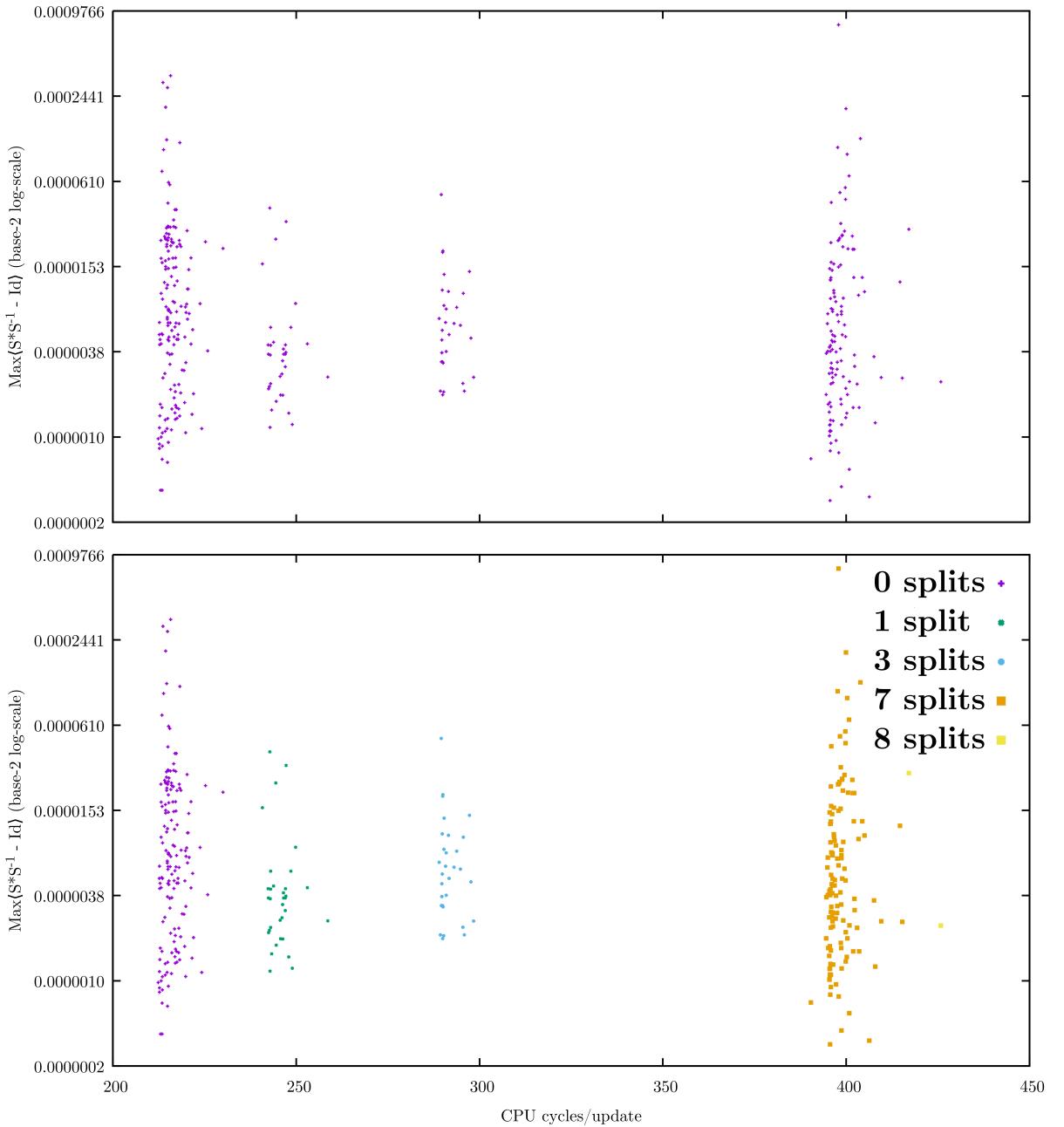


Figure 5.7: Performance stratification caused by update splitting.

To look a bit closer at the big cluster in the middle we selected only the update-chains containing nine rank-1 updates and only plotted the ones that passed with the “Splitting” kernel. This is shown in the left panel of Figure 5.7. In this case the vertical stripes are formed by splits, where the spacing between the stripes correspond to how many times the kernel had to split an update before it could be successfully applied. Not every possible number of splits occurs, as can be seen in the right panel of Figure 5.7. Here we separated the points by the number of splits. In the case of nine rank-1 updates, only 0, 1, 3, 7 and 8 splits occur, hence difference in the spacing of the stripes.

6 Classification of all the kernels

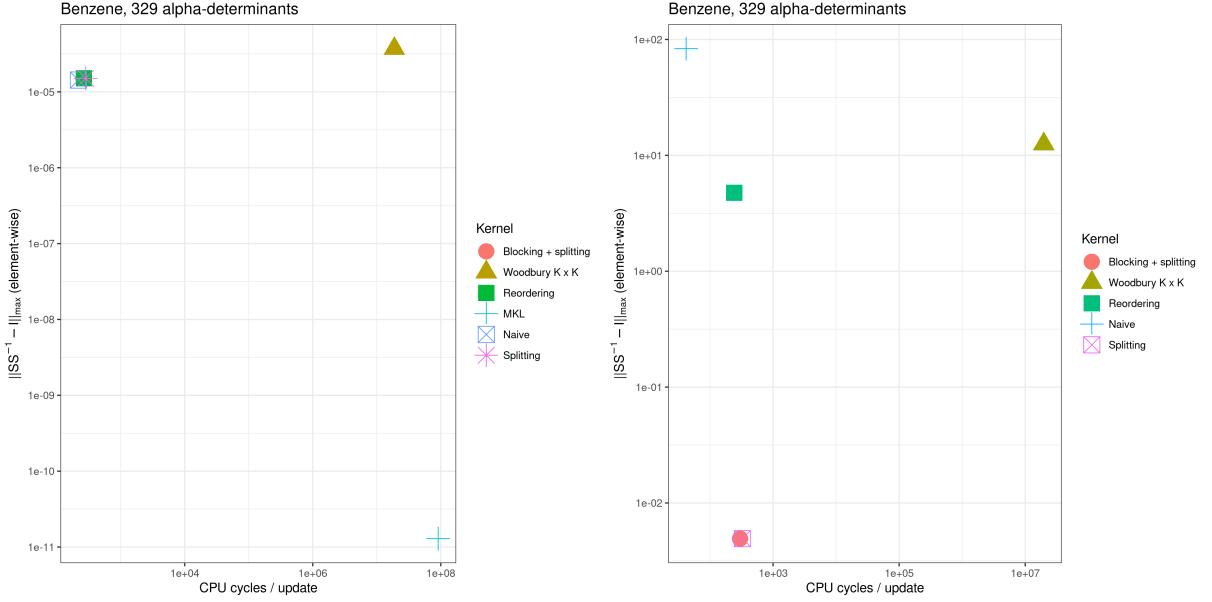


Figure 6.1: Numerical accuracy versus performance

To have a good idea of the numerical accuracy and performance of each of the kernels we have included Figure 6.1. It is similar to Figure 5.6 but for each kernel we averaged the results for the numerical accuracy along the vertical axis, the element-wise max-norm of ρ , and took the median of the number of CPU cycles per rank-1 update along the horizontal axes. This results in an easy to read overview for each kernel. Ideally the best kernel resides in the lower-left corner of the square and the worst possible one in the upper-right corner.

What is immediately apparent when looking at the left panel, is that even though the inverses computed directly with Intel’s MKL are the most accurate (lowest max-norm of ρ), it is also the most costly. Similarly, even though the Naïve kernel is the fastest, it is also the least accurate in cases where there are more than one rank-1 update.

To have a better resolution in the part where the remaining kernels are clustered together the MKL kernel has been removed in the second panel. In the the lower-left corner we can see the two kernels Splitting and Blocking+splitting (pink square and red dot). Their performance is similar to the Reordering kernel (green square) but have a better numerical accuracy of at least one order of magnitude. This is good news because it should help with the issues mentioned in Section 5.2.1 and 5.2.2.

For some reason the Woodbury $K \times K$ kernel performs relatively bad in terms of accuracy and performance.

6.1 Final recommendations

Summarising the results from Table 5.2 and Figure 6.1 we can conclude with the following recommendations

Single rank-1 updates When there are only single rank-1 updates we don’t have to worry about the relatively low numerical accuracy of the Naïve kernel because this is only due to the

possible singular behaviour induced by iterative process in multi rank-1 updates. In this case the **Naïve** will give you the best performance overall.

All other cases In all other cases the Blocking+Splitting kernel is the best option. It has the same numerical accuracy as the Splitting kernel but with the benefit of being slightly faster due to its higher arithmetic intensity.

7 Inclusion in Quantum Monte Carlo Kernel Library

Since this work has been funded by and executed in the context of the TREX Centre of Excellence in HPC for Quantum Chemistry[3], all the computational kernels that are presented in this report (except the kernel outlined in Section 3.2) are included in the TREX Centre of Excellence Quantum Monte Carlo Kernel Library[2] (TREX-CoE/QMCKl). The repository where the QMCKl code and documentation is stored can be found on <https://github.com/TREX-CoE/qmckl>.

The kernels themselves as well as the code, documentation and regression testes can be found in an ‘Org-mode’ formatted file can be found the `org/` subdirectory at https://github.com/TREX-CoE/qmckl/blob/master/org/qmckl_sherman_morrison_woodbury.org.

The HPC versions of these kernels can be found at <https://github.com/TREX-CoE/qmckl/tree/master/org/hpc>.

8 Preliminary work on $K \times K$ Woodbury on the GPU

There are some preliminary results after a first attempt on porting the $K \times K$ Woodbury kernel to GPU architectures. Since this is ongoing work it was not included in Section 4, nor yet in the QMCKl. The ultimate goal would be to have a GPU implementation that performs faster, on a general GPU architecture, under typical workloads, than on a (multi-core) CPU. Preferably it uses only open standards like OpenMP offloading, OpenACC, OpenCL or SYCL to increase portability and prevent a “vendor lock-in”. If and when such a version is realised, it can be added to the QMCKl.

The starting point has been the implementation of Algorithm 5 on the CPU. We used OpenMP to offload the data –and the loops that could not be written in terms of standard BLAS/LAPACK-calls– to the GPU. The parts that are expressed as standard BLAS/LAPACK calls have been replaced with calls to NVIDIA’s cuBLAS/cuSOLVER libraries.

The the GPU specific hard- and software used for the performance measurements are the same as mentioned in Section 5.2 with the addition of a NVIDIA GPU using the Ampere architecture (GA102) with 10GB of device memory and CUDA Compute Capability 8.6. The GPU-ready code was compiled with NVIDIA’s `nvc/nvc++` compilers and profiled/analysed with NVIDIA Nsight Systems/Compute and NVIDIA Tools Extension (NVTX) included in the NVIDIA HPC SDK version 22.7 (July 2022).

8.1 GPU warm-up effects

In Figure 8.1 we show the average number of CPU cycles as a function of the matrix-size for different update loads and repetition number. The number of rank-1 updates for a particular workload and matrix size is $N_{\text{updates}} = l \times M$, where l is the load and M is the size of the matrix. The load and matrix sizes are chosen such that N_{updates} is always an integer. The sizes and loads that were used are $M \in \{32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$ and $l \in \{0.25, 0.50, 0.75\}$.

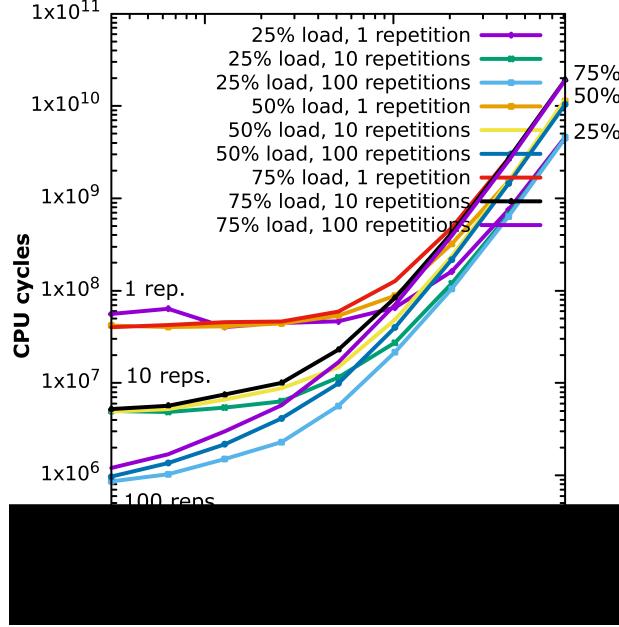


Figure 8.1: GPU warm-up effects for smaller matrices.

The repetition means how many times the kernel was executed for a specific matrix size and update load. We chose 1, 10 and 100 repetitions. The final number of CPU cycles have been divided by the number of repetitions. It is clear that the number of repetitions matters for smaller matrices. The larger the number of repetitions, the lower the averaged number of CPU cycles. Around matrix sizes of 2048 the warm-up effect disappears and it is the update load that determines the final performance.

A possible explanation could be that for larger matrices the execution time of the kernel is already of the same order of magnitude as the warm-up itself, which happens too fast compared the timescales of the execution of the larger ($> 1024 \times 1024$) matrices.

8.2 Multi-threaded CPU vs GPU

Finally, in Figure 8.2 we compare the performance curves of a full matrix inversion using MKL with 10 OpenMP threads on the CPU, $K \times K$ Woodbury with 10 OpenMP threads on the CPU and $K \times K$ Woodbury on the GPU. The curves for the 25% workloads are in blue, the 50% in green and the 75% in red. For small matrices the GPU kernel is always slower, but performs more or less the same for all workloads. MKL inversion looks to be the best for all workloads. Up until matrices of size 512 the Woodbury CPU kernel seem to be doing best for workloads up to 50% and performs equivalent to MKL at 75%. From matrices of size 1024 and upwards the Woodbury GPU kernel starts to be at comparable performance with the CPU kernels. For very large matrices of size around 8196 the Woodbury GPU kernel is the fastest for 25% and 50% workloads, but MKL is faster for 75%.

Kernel	Time (s)	Time (cc)	Speed-up
Sequential	454.352	1.996482×10^{11}	1
OpenMP, 10 threads	108.040	4.714318×10^{10}	4.23
GPU	102.776	3.395985×10^{10}	5.88

Table 8.1: $K \times K$ Woodbury for 1 repetition, matrix size: 16384×16384 , 4096 updates (25%)

From Table 8.1 it is clear that the $K \times K$ Woodbury kernel on the GPU in its current state is

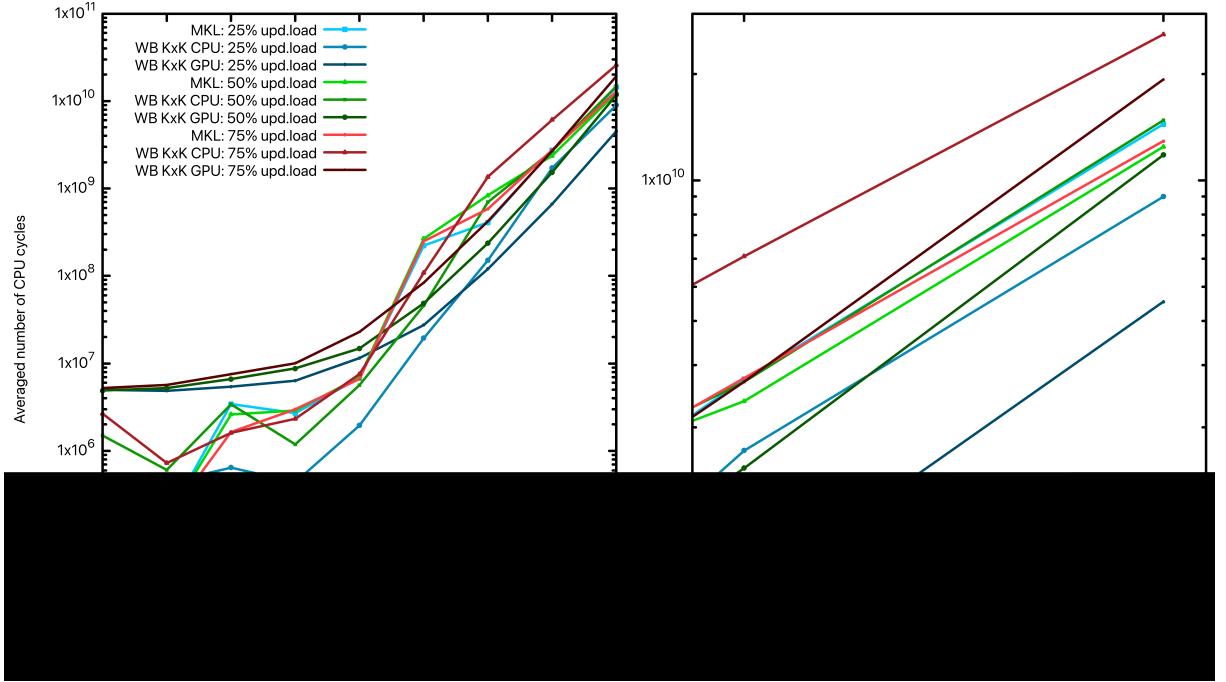


Figure 8.2: GPU warm-up effects.

only marginally faster than the OpenMP threaded version, running on 10 threads. More analysis with profiling tools like NSight Systems/Compute and MAQAO is needed to find out if

1. we can model/predict what acceleration we could reasonably expect from a kernel like this on GPU architectures,
2. in order to know how far away we are from optimal performance,
3. we are using open programming models like OpenMP offloading, OpenACC correctly,
4. if there are problems with their implementation in the used compilers,
5. we need to resort to lower-level programming using C++ language extensions like CUDA, OpenCL or SYCL.

Appendices

Appendix A Zero-padding

Lets define the size of a vector register as v . Then for general $k \times l$, $l \times m$ matrices A, B the amount of zero-padding p on the inner/outer dimension l of A/B is equal to

$$p = v - (l \bmod v) \quad (28)$$

as long as l is not an integer multiple of v . In that case $p = 0$. So we have that $0 \leq p \leq v - 1$. The padded dimension n is then

$$n = l + p \quad (29)$$

We can then define the $l \times n$ padding-matrix $P_{l,n}$ as the augmented matrix

$$P_{l,n} := (I_l | 0_{l,p}) \quad (30)$$

where I_l is the $l \times l$ identity-matrix and $0_{l,p}$ is the $l \times p$ null-matrix. The matrix $P_{l,n}$ has the properties such that

$$P_{l,n} P_{n,l}^\top = I_l \quad (31)$$

and

$$P_{n,l}^\top P_{l,n} = \begin{bmatrix} I_l & 0_{l,p} \\ 0_{p,l} & 0_{p,p} \end{bmatrix} \quad (32)$$

Properly padded matrices can then be simply obtained by multiplying $P_{l,n}$ with $A_{k,l}$ to pad the columns in a case column-major layout is used.

$$\tilde{A}_{k,n} = A_{k,l} P_{l,n} \quad (33)$$

or $P_{n,l}^\top$ with $A_{l,m}$

$$\tilde{A}_{n,m} = P_{n,l}^\top A_{l,m} \quad (34)$$

to pad the rows in case a row-major layout is used.

A.1 Sherman-Morrison with zero-padding

We can now use this to add padding to the Sherman-Morrison formula to make it more efficient in SIMD enabled CPUs. We go from using matrices S , S^{-1} and u to using matrices \tilde{S} , \tilde{S}^{-1} and \tilde{u}

$$(S + uv^\top)^{-1} = S^{-1} - \frac{S^{-1}uv^\top S^{-1}}{1 + v^\top S^{-1}u} \quad (35)$$

$$\downarrow \quad (36)$$

$$(\tilde{S} + \tilde{u}\tilde{v}^\top)^{-1} = \tilde{S}^{-1} - \frac{\tilde{S}^{-1}\tilde{u}\tilde{v}^\top \tilde{S}^{-1}}{1 + v^\top \tilde{S}^{-1}\tilde{u}} \quad (37)$$

$$\Downarrow \quad (38)$$

$$(P^\top S + P^\top uv^\top)^{-1} = S^{-1}P - \frac{S^{-1}PP^\top uv^\top S^{-1}P}{1 + v^\top S^{-1}PP^\top u} \quad (39)$$

for padding added to the columns of S and u and to the rows of S^{-1} . This insures that

$$\tilde{S}^{-1}\tilde{S} = S^{-1}PP^\top S \quad (40)$$

$$= S^{-1}IS \quad (41)$$

$$= S^{-1}S \quad (42)$$

A.2 Woodbury with zero-padding

For the Woodbury formula we proceed in the same way. The matrices S , S^{-1} and U are replaced by \tilde{S} , \tilde{S}^{-1} and \tilde{U}

$$(S + UV)^{-1} = S^{-1} - S^{-1}U(I + VS^{-1}U)^{-1}VS^{-1} \quad (43)$$

$$\downarrow \quad (44)$$

$$(\tilde{S} + \tilde{U}V)^{-1} = \tilde{S}^{-1} - \tilde{S}^{-1}\tilde{U}(I + V\tilde{S}^{-1}\tilde{U})^{-1}V\tilde{S}^{-1} \quad (45)$$

$$\Downarrow \quad (46)$$

$$(P^\top S + P^\top UV)^{-1} = S^{-1}P - S^{-1}PP^\top U(I + VS^{-1}PP^\top U)^{-1}VS^{-1}P \quad (47)$$

References

- [1] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009. Santa Clara, USA. ISBN 630813-054US.
- [2] The QMCKl library for software developers and HPC experts. <https://github.com/TREX-CoE/qmckl>, 2020.
- [3] TREX Centre of Excellence in HPC for Quantum Chemistry. <https://trex-coe.eu/about-trex-hpc-centre-excellence-quantum-chemistry>, 2020.
- [4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [5] F. Coppens. Extracted QMC=Chem datasets that were used in producing the results in this report. `rsync://trex@trex-share.univ-tlse3.fr:uvsq/`, 2021.
- [6] P. Maponi. The solution of linear systems by using the sherman–morrison formula. *Linear Algebra and its Applications*, 420(2):276–294, 2007.
- [7] A. Scemama. Quantum Monte Carlo for Chemistry. http://qmccchem.ups-tlse.fr/index.php/Quantum_Monte_Carlo_for_Chemistry @_Toulouse, 2015.
- [8] J. T. Slagel. The sherman morrison iteration. Master's thesis, Virginia Polytechnic Institute and State University, 2015.