HPC Sherman-Morrison-Woodbury kernels for QMC applications

François Coppens* Pablo Oliveira † Eric Petit ‡

November 29, 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 7
3.4 Splitting the difference: Slagel's method 8
3.5 All at once: the Woodbury matrix identity 9

4 HPC versions **10**
4.1 SIMD and zero-padding (maybe put math in an annex and just say we can padd.) 10
4.2 Manual loop-unrolling: 2×2 and 3×3 WB and the Blocking kernel 11
4.3 Explicit array dimensions and loop bounds 13
4.4 Aligned and restricted dynamic arrays 13
4.5 Template to generate kernels for specific sizes 13

5 Experiments with QMC=Chem **13**
5.1 Problem 1: Numerical accuracy with large number of determinants 14
5.2 Problem 2: Possibly unnecessary calls to LAPACK 14
5.3 Problem 3: Arithmetic intensity 14
5.4 Dataset extraction 14
5.5 Measurement procedure 15
5.6 Numerical accuracy 16
5.6.1 329 α-determinants 17
5.6.2 15784 α-determinants 19
5.7 Performance 19
5.7.1 Performance per rank-1 update 19
5.7.2 Performance stratification 20
5.8 Final classification of all the kernels 21

6 Discussion and recommendation **21**

*UVSQ
†UVSQ
‡Intel Corp.

7 Inclusion in Quantum Monte Carlo Kernel Library **21**

A Sherman-Morrison with zero-padding **23**

B Woodbury with zero-padding **24**

References **25**

TODO

- Section 2 - merge with section 5 to avoid forward references, simplify its introduction. Explain the dataset extraction.
- Section 3 - Missing the introduction to the section. Explain why we are considering these methods, Method 3.5 is missing some accompanying text. It should be clear what each method is bringing and why it was chosen.
- Section 4 - Reorganise, simplify padding merge with constant + alignment and separate optimisations for the blocked case. Include vectorisation figures after and before. Highlight the HPC aspects and why it improved the performance.
- Section 2+5. Describe the architectures used for the measurements. Describe the importance of testing in real datasets from QMC=Chem. Compare and present the advantages of each method.
- Merge 5.4 and 6. Fill 6.
- Small extra part (independent) to summarise the GPU results ?

1 Aims and contributions

This work has been executed in the context of the TREX-CoE[add_ref]. 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[add_ref] 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[add_ref].

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 the problem with the current implementation will be pointed out, as well as how it arises in Section 2.2.

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

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

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

where $\mathbf{R} = \mathbf{r}_1, \ldots, \mathbf{r}_N$, $\mathbf{R}_{\uparrow}$ 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

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

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

$$E_{\text{kin}} \propto -\sum_{i=1}^{N} \nabla^2_{\mathbf{r}_i} \Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N) \tag{3}$$

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

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

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[ref here] or MKL[ref here], 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

$$U = uv^\top \tag{5}$$

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 $\widetilde{A}$

$$\widetilde{A} = A + U = A + uv^\top \tag{6}$$

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

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

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 $\widetilde{A}^{-1}$ can then be computed from the old inverse A^{-1} with the following formula

$$\widetilde{A}^{-1} = \left(A + uv^\top\right)^{-1} = A^{-1} - \frac{A^{-1}uv^\top A^{-1}}{1 + v^\top A^{-1}u} \tag{8}$$

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

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

such that

$$\widetilde{A} = \sum_{k=0}^{K} A_k = A_0 + \sum_{k=1}^{K} u_k v_k^\top \tag{10}$$

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

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

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

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 $\widetilde{A}^{-1}$ is guaranteed to exists, 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

$$0 < \beta \ll 1 \tag{12}$$

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 values was also used during the experiments we did. Then for a given β, A_{k-1}^{-1} and u_k we stop updating when

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

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 exists. 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.

The method we used to measure the success rate is straight forward. If we can define the following residual matrix

$$\rho := \widetilde{A}^{-1}\widetilde{A} - I \tag{14}$$

we can use ρ to specify an error on $\widetilde{A}^{-1}$ in the same way that we use the break-down parameter β. As before we define the tolerance

$$0 < \tau \ll 1 \tag{15}$$

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

$$\|\rho\|_{\max} < \tau \tag{16}$$

Whenever this inequality holds the numerical error on $\widetilde{A}^{-1}$ 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.

3.1 An naïve approach to iterative Sherman-Morrison

This method is inspired by Eq. (15) in [1]

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

It omits computing the solution and auxiliary solution vectors x_k and $y_{k,l}$. During the course of applying the chain of rank-1 updates the denominators in Eqn. (17) are checked with the condition Eqn. (13).

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: $k \leftarrow k + 1$;

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

$$A\mathbf{x} = \mathbf{b} \tag{18}$$

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[1] 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

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

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 $\left|1 + v_l^\top A_k^{-1}u_l\right|$ 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 finally we are not interested in the solution $\mathbf{x}$ but merely in the inverse 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:
  $d_k \leftarrow \left|1 + v_0^\top y_{l-1,k}\right|$;
  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 $\{y_{l,k}\}$:
    $y_{l,k} \leftarrow y_{l-1,k} - \frac{v_l^\top y_{l-1,k}}{1+v_l^\top y_{l-1,l}}y_{l-1,l}, \quad k \in \{l+1,\ldots,K\}$;

3.3 Reordering the updates

This method is the one currently used in QMC=Chem. It is an improvement of the "Naïve" method stated in Section 3.1, where an update u_k is kept back in a delay queue
This method is the one currently used in QMC=Chem. It is an improvement of the "Naïve" method stated in Section 3.1 where, when Condition (**??**) is triggered for some update u_k along

the chain of k-updates, the update is sent to 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 updates. If not, it is also send to the delay-queue. Then the next update is evaluated, etc., 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. The algorithm is implemented as a recursive function in C.

This method is equivalent to the one that has been used in QMC=Chem.

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: $\{u_l, v_l\} \leftarrow \varnothing$;
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 Splitting the difference: Slagel's method

This method based on the nave approach of Section 3.1 and improved with an implementation of Joseph Tanner Slagel's splitting method[2].

Remember the example from Section 2.2 where the positions of two electrons are exchanged. Even though this cannot be done one-by-one imagine that instead of simply exchanging them we move electron-1 half-way in the direction of electron-2, clock-wise, and electron-2 half-way to electron one, also clock-wise. This is perfectly acceptable. Once this is done we do it again.

Now electron-1 and electron-2 positions are switched, we did it incrementally and we didn't cause a singular intermediate Slater determinant. This is how one can conceptualise the idea of Joseph T. Slagel. By splitting the update that would cause a singular matrix, apply one half AND sending the other halve to a delay-queue we can avoid the pathology of applying rank-1 updates iteratively.

It can be proved that using this method it takes 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.

PUT PROOF HERE

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: $\{u_l, v_l\} \leftarrow \varnothing$;
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 it is also possible to generalise the SM formula for rank-k updates. This is the Woodbury matrix identity (WB) and in fact SM is the special case for where the update matrix U is of order rank-1. This is its most general form

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

where $U : N \times K$, $C : K \times K$ and $V : K \times N$ are conformable matrices. Notice that

$$D := C^{-1} + VA^{-1}U \tag{21}$$

is at most $K \times K$. For example, when calculating the electronic structure and ground state properties of the compound Benzene with QMC=Chem, 75% of the time the number of updates that need to be applied to the inverse Slater-matrix is not larger than six, so $K = 6$, whereas for Benzene $N = 21$!

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

$$D = 1 + v^\top A^{-1} u \tag{22}$$

and because $v^\top A^{-1} u$ is scalar, $D^{-1} = 1/\left(1 + v^\top A^{-1} u\right)$, so we recover the SM formula.

We compute $\widetilde{A} = A^{-1} - CD^{-1}E$, where $C = A^{-1}U$, $D = 1 + VC$ and $E = VA^{-1}$.

Algorithm 5: The "$K \times K$ Woodbury" kernel

Data: A^{-1}, $\dim A^{-1}$, $\det A^{-1}$, K, U, V, β
Result: $\widetilde{A}$, $\det \widetilde{A}$
compute: $C \leftarrow A^{-1}U$;
compute: $D \leftarrow 1 + VC$;
compute: $E \leftarrow VA^{-1}$;
compute: $\det D$ (by LU-decomposition);
if $|\det D| < \beta$ **then**
  exit;
update: $\det A \leftarrow \det A \times \det D$;
compute: D^{-1} (from earlier LU-decomposition);
compute: $A^{-1} \leftarrow A^{-1} - CD^{-1}E$;

4 HPC versions

4.1 SIMD and zero-padding (maybe put math in an annex and just say we can padd.)

Many loops cannot be completely vectorised when the loop-bounds are not integer multiples of the CPU's vector length. In that case tail-loops need to be inserted and this will have a negative impact on performance. It is then sometimes better to add a small amount of unnecessary work by padding vectors and matrices with extra zeros so that their sizes and the loop-bounds are integer multiples of the vector size and tail-loops are eliminated. In that case, the SIMD registers are used and the computation as a whole is faster, even if slightly more work has been done.

The amount of zero-padding depends on the vector length of the CPU. For example for an Intel CPU with AVX2 support the vector length is 256 bits, which is equivalent to 4 64-bit double precision floating point numbers or 8 32-bit single precision floating point numbers. For AVX512 these numbers double. Lets define the vector length as

$$v := \text{vector length of the CPU} \tag{23}$$

Then for general $k \times l$, $l \times m$ matrices A, B the amount of zero-padding

$$p := \text{amount of zero-padding} \tag{24}$$

on the inner– and outer dimension l of A, B is equal to

$$p = \begin{cases} v - l \bmod (v), & l \bmod (v) \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{25}$$

such that

$$0 \leq p \leq v - 1 \tag{26}$$

We then set the padded dimension

$$n = l + p \tag{27}$$

We can then define the $l \times n$ padding-matrix $P_{1l,n}$ as the augmented matrix

$$P_{l,n} := (I_l |\, 0_{l,p}) \tag{28}$$

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

$$P_{l,n} P_{n,l}^\top = I_l \tag{29}$$

and

$$P_{n,l}^\top P_{l,n} = \begin{bmatrix} I_l & 0_{l,p} \\ 0_{p,l} & 0_{p,p} \end{bmatrix} \tag{30}$$

We can then construct zero-padded matrices using $P_{l,n}$ to pad the columns of $A_{k,l}$

$$\widetilde{A}_{k,n} = A_{k,l} P_{l,n} \tag{31}$$

or $P_{n,l}^\top$ to pad the rows of $A_{l,m}$

$$\widetilde{A}_{n,m} = P_{n,l}^\top A_{l,m} \tag{32}$$

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

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.

Algorithm 6: The "Blocking" kernel

Data: A^{-1}, $\dim A^{-1}$, $\det A^{-1}$, K, U, V, β
Result: $\widetilde{A}$, $\det \widetilde{A}$
first initialisation:
number of later updates: $L \leftarrow 0$;
later updates queue: $\{u_l, v_l\} \leftarrow \varnothing$;
if $K = 4$ **then**
  call WB 2×2 with the **first** 2 rank-1 updates;
  if *WB* 2×2 *fails* **then**
    call Splitting-kernel^1 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-kernel^1 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-kernel^2 using A^{-1}_{K-L}, $\dim A^{-1}_{K-L}$, $\det A^{-1}_{K-L}$, L, $\{u_l, v_l\}$;
  exit;

The algorithm looks as follows:

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-kernel[1] 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-kernel[1] 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-kernel[1] 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-kernel[2] using A^{-1}_{K-L}, $\dim A^{-1}_{K-L}$, $\det A^{-1}_{K-L}$, 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 loop bounds

For the case of Benzene computed on Intel AVX2 we have hard coded the matrix and padding sizes in the source code to assist the compiler to maximise loop-vectorisation.

4.4 Aligned and restricted dynamic arrays

All the arrays that are passed are memory aligned and access is restricted to prevent aliasing.

4.5 Template to generate kernels for specific sizes

For the HPC version of the kernels there is a Python script that generates the C-source code for specific system sizes and SIMD architectures. This allows the C-compiler to always know all the loop-bounds at compile time, thereby ensuring that all loops are vectorised.

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. In the version we used to test and extract the datasets a form of iterative SM was already being used. Before the individual determinants are updated after an electron move they are ordered first in such a way that the next determinant only differs by the smallest possible number of columns, molecular orbitals (MOs), from the previous one. Then the first inverse in the sequence is computed fully with LAPACK. The next one, often differing only one column and therefore only one rank-1 update from the previous one, is computed using SM. Then the next one, etc. until the end of the chain is reached.

5.1 Problem 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.

5.2 Problem 2: Possibly unnecessary calls to LAPACK

Due to Problem 1, 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.

5.3 Problem 3: Arithmetic intensity

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 **??**.

5.4 Dataset extraction

We tested our kernels on datasets 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 an electron move, when the MOs are being updated and the determinants in Eqn. 1 are updated as well. Additional Fortran statements have been added to the code to extract and save the following data

- the number of the determinant in the chain, starting from 1 and omitting numbers $i \times N_{\text{dets}},\ i = 0, 1, \ldots$ 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.

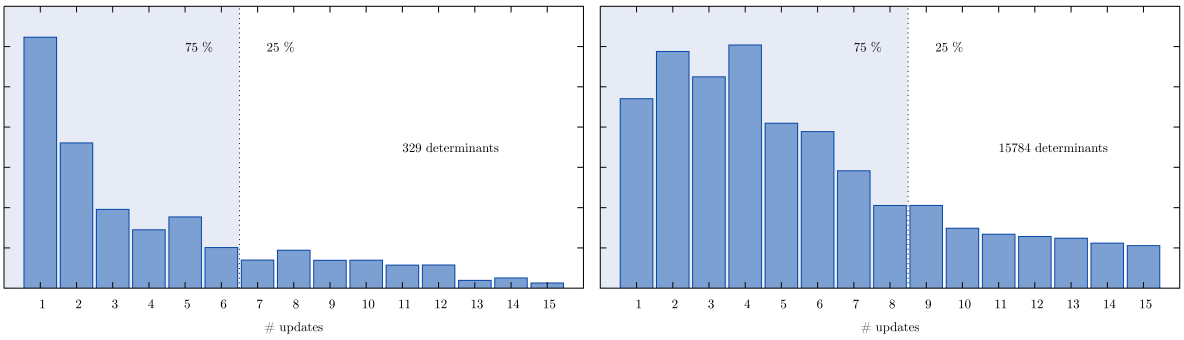

Figure 5.1: Distribution of the occurrence frequency for each number of rank-1 updates that can occur. The blue area represents the 3$^{\text{rd}}$ quartile of the distribution. For the 329-determinants case, 75% of the time updates consist of 1–6 rank-1 updates. For the 15784-determinants case updates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

5.5 Measurement procedure

Each determinant that is updated can have 1 to 15 rank-1 updates. We call this an **update cycle**. For each update cycle we read the following data from the dataset (except D):

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

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 $\left|1 + v_k^\top A_{k-1}^{-1} u_k\right| > \beta$, 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^{-1}_{copy} 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^{-1}_{copy} 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`

5.6 Numerical accuracy

The 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

$$\text{Fail rate} = \frac{\#\text{ of FAILS}}{\#\text{ of PASSES} \quad + \quad \#\text{ of FAILS}} \times 100\% \tag{33}$$

5.6.1 329 α-determinants

In Table 5.1 the fail rates are shown for 10384 determinants for the 329 determinant 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, 10384 determinants 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. The 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.

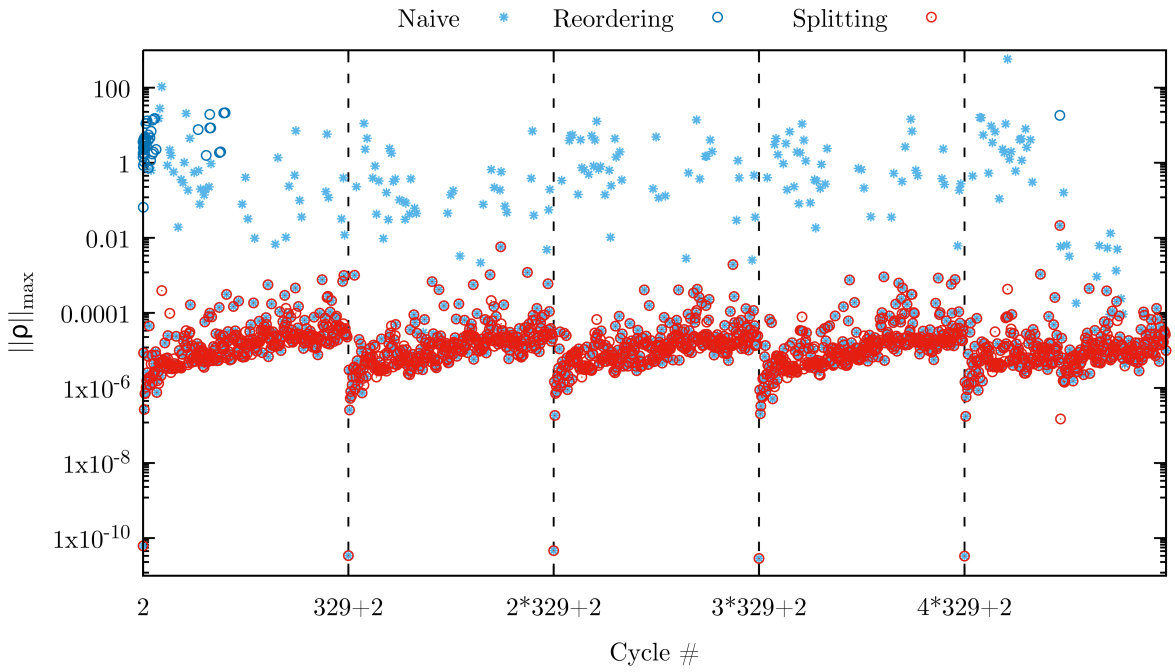

Figure 5.2: Numerical drift, steadily increasing from the beginning to the end of the 329 determinant update chain.

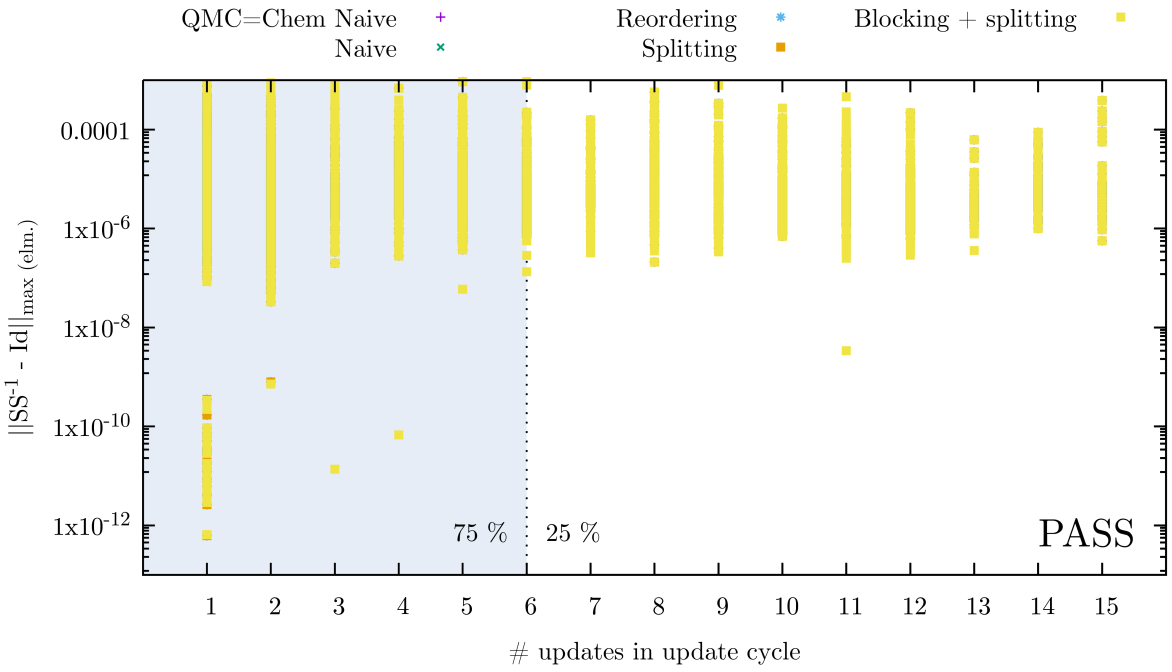

Figure 5.3: The influence of the number of rank-1 updates to numerical accuracy for various kenrnes.

5.6.2 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 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.

5.7 Performance

5.7.1 Performance per rank-1 update

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 3$^{\text{rd}}$ 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 occurences are clustered around 0–2 splits and take an average of about 300 CPU cycles per update.

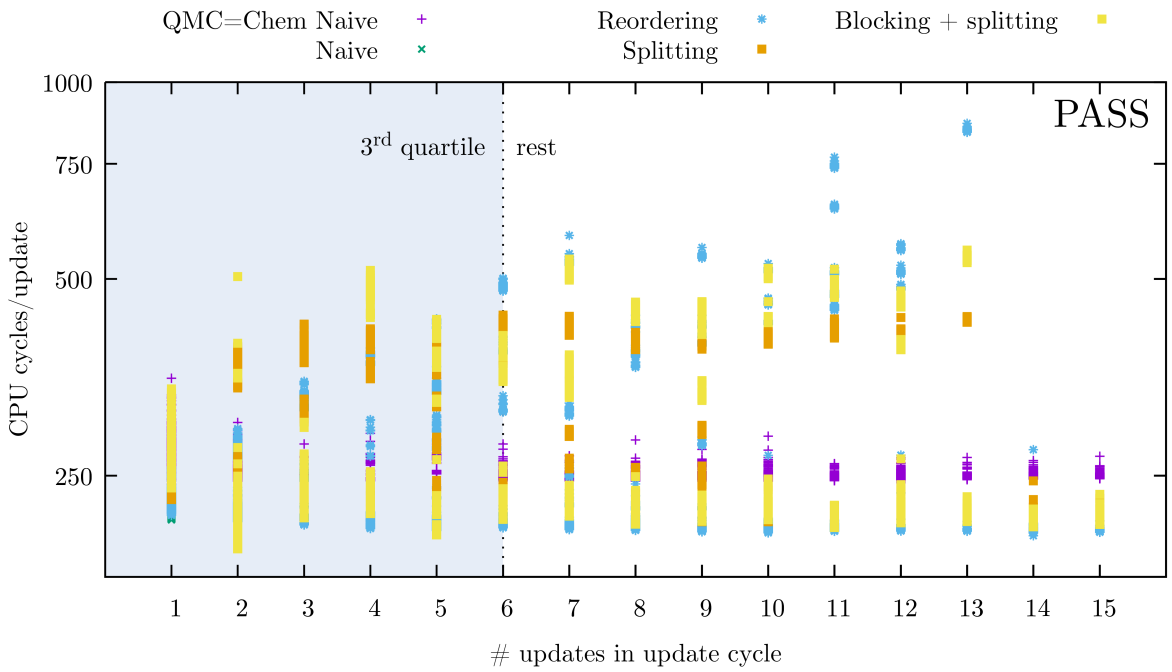

Figure 5.4: Performance in CPU-cycles per rank-1 update for various kernels.

5.7.2 Performance stratification

When you look at Fig. 5.6 of everything together, all kernels, all updates, pass and fail, it you get a good overview of the whole space of possibilities but you cannot see well what is going on for each individual kernel. It is still somewhat instructive because you can see that most of the failures come from Naive and QMC=Chem Naive (covered by Naive). That was to be expected.

2. In Fig. 5.7 we are only looking at the cycles that passed. It is already a bit better but is still a bit messy. Not sure this plot is much better than the first one. Maybe we should not include it. 3. In the third plot I focused on only the passed updates with 9 cycles. This is already much cleaner and gives a better idea of what the kernels are doing. And it is here were you can clearly see some stratification of the clusters for each individual kernel, except for the 2 Naive ones and LAPACK.
4. To understand the stratification I thought of zooming in on one kernel to see if it had something to do with splitting. So the next plot only shows the Pareto plot for all passed 9-update cycles of the splitting kernel. Here you can see clearly 4 distinct bands. First I thought this was linked to the maximum number of splits for a given 9-update cycle. But this is not the case because the maximum number of splits is 8 and there are not 9 bands (one extra for 0 splits). Then I thought it could be related to the number of recursions the kernel calls itself, but this was also not the case because the maximum number of recursions is only 1x: all splits happen in the first pass and get completed in the second pass (for 9-update cycles). Finally it is related to the number of splits, but not all number of splits happen. In fact only 0, 1, 3, 7 and 8 splits happen for 9-update cycles. So when you then split out the data for these number of splits you get the final plot. 5. In the last plot you can clearly see that the bands correspond to each bin of number of splits. There are actually 5 bands, but the band for 8-splits only contains 2 points and is very close to the points for the 7-split band. If there were more points I am sure they. would form a band as well. I am pretty convinced this also explains the bands in the performance plots for splitting @Eric P. mentioned 2 weeks ago when I was at UVSQ. For the Later-kernel these bands must be related to the number of recursions because it doesnt do splitting. For the Blocked kernel it is also splitting but the distribution of the number of splits

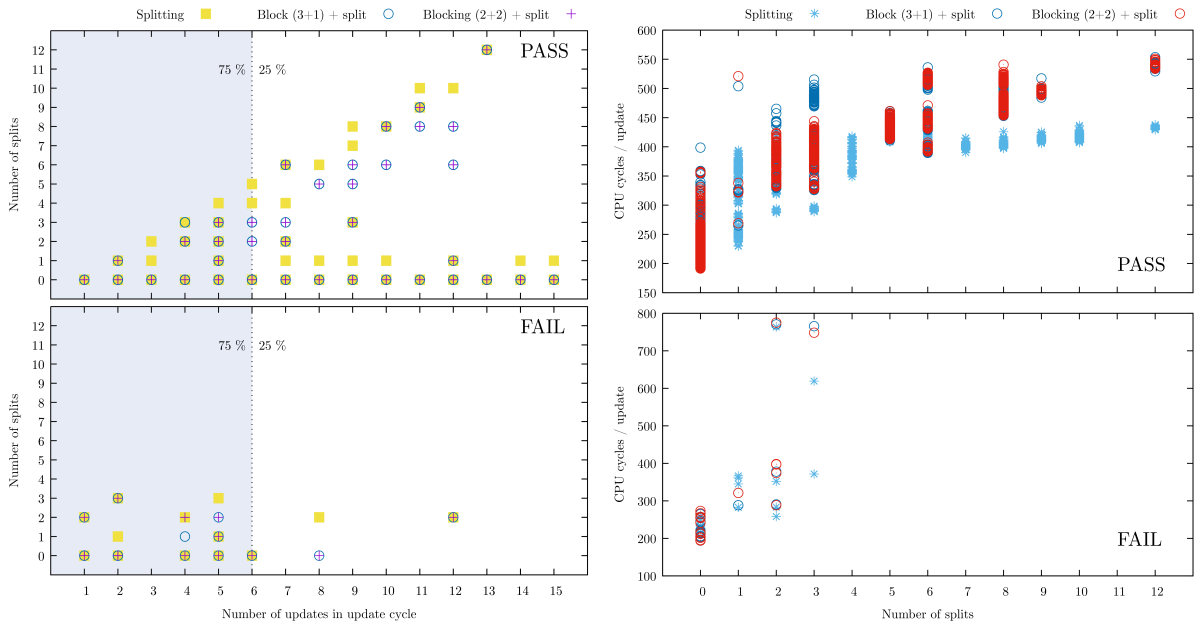

Figure 5.5: Distribution of the occurrence frequency for each number of rank-1 updates that can occur. The blue area represents the 3rd quartile of the distribution. For the 329-determinants case, 75% of the time updates consist of 1–6 rank-1 updates. For the 15784-determinants case updates consist of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

is modulated by the fact some Woodbury blocks of 3 succeed where there would otherwise be a split. If it is needed I can do the same analysis for the Blocked kernel. (

5.8 Final classification of all the kernels

Fig. 5.8

6 Discussion and recommendation

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, 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 (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`

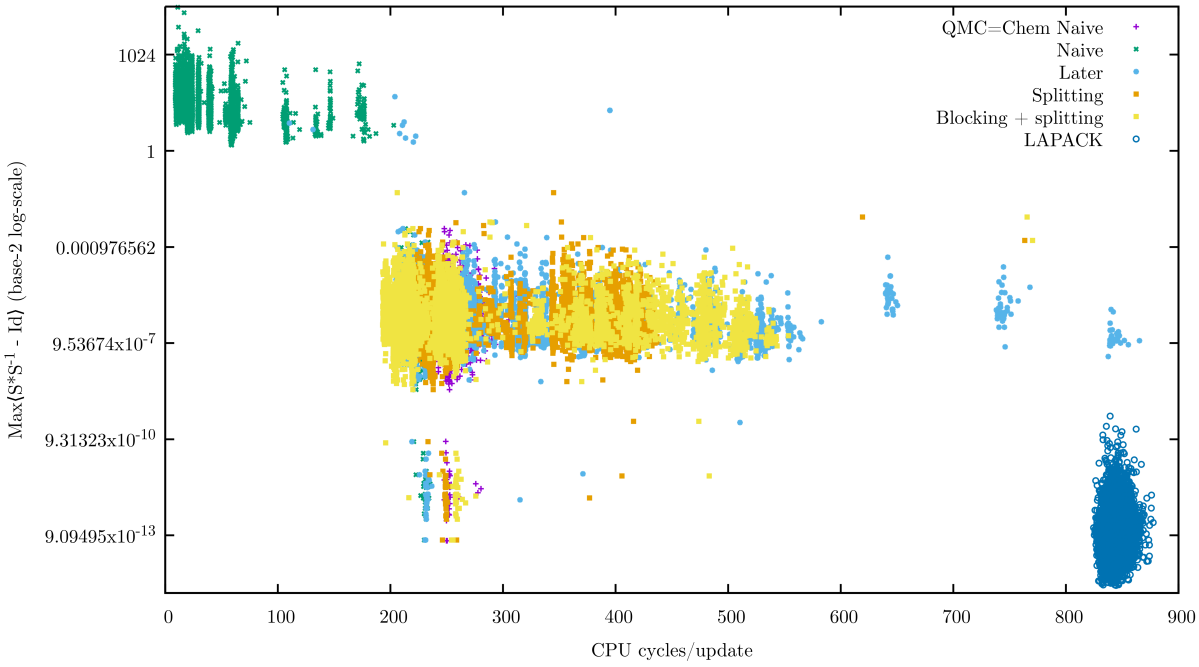

Figure 5.6: The accuracy ($||SS^{-1} - I||_{\max}$) vs. the performance for all tested kernels, above and below the acceptance tolerance.

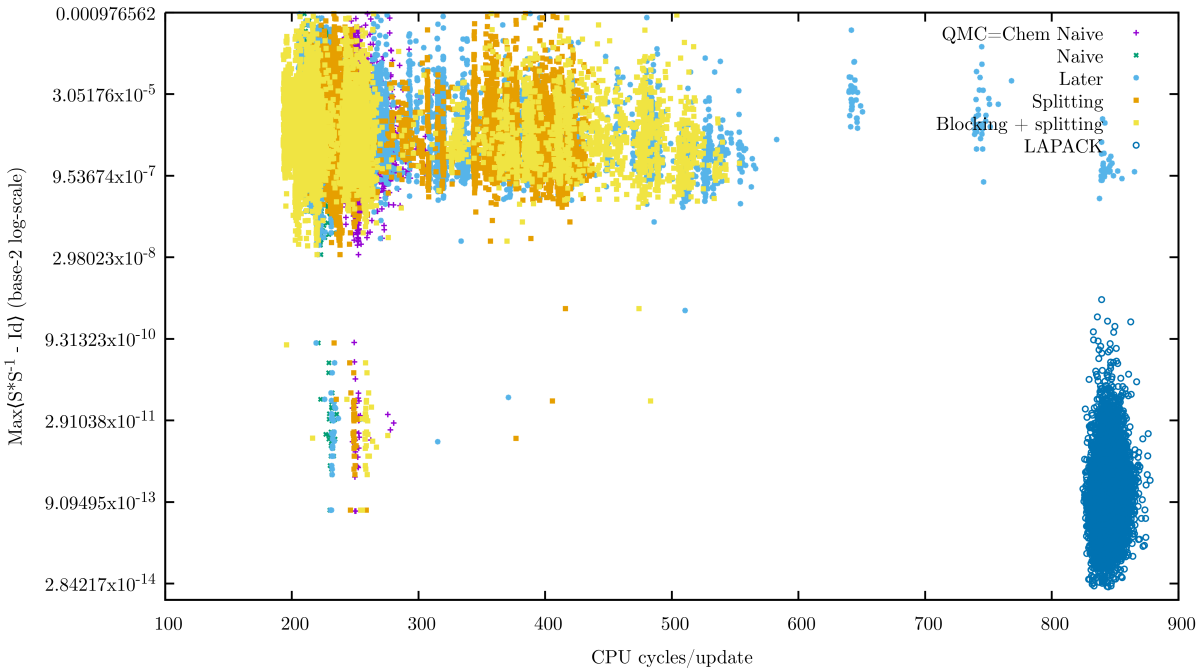

Figure 5.7: The accuracy ($||SS^{-1} - I||_{\max}$) vs. the performance for all tested kernels, above and below the acceptance tolerance.

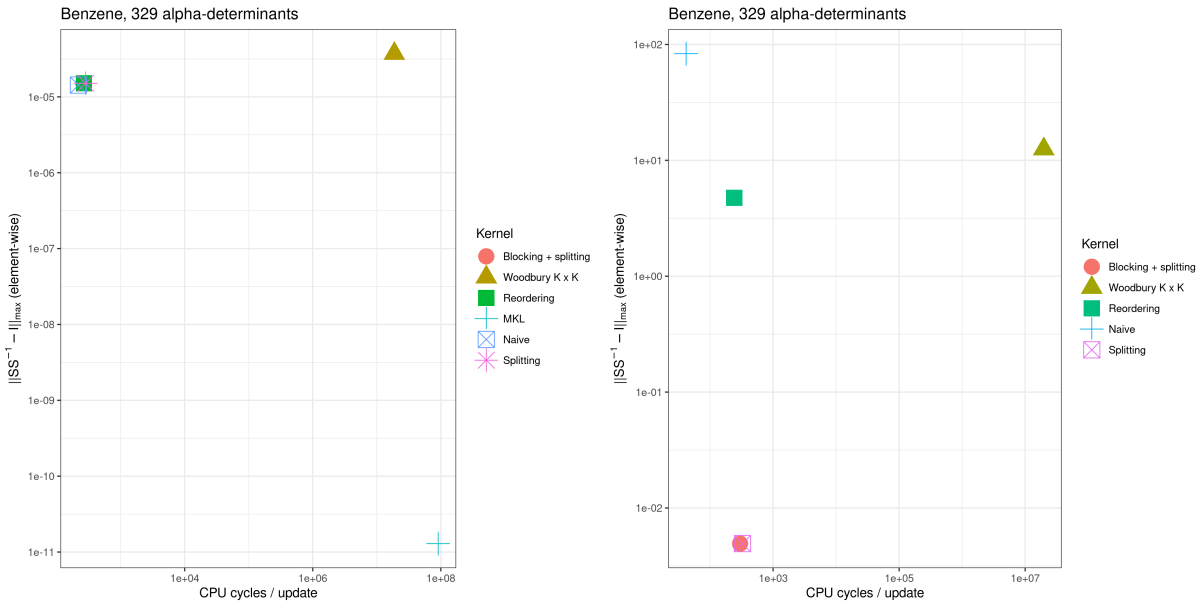

Figure 5.8: kjkjkj

A 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 $\widetilde{S}$, $\widetilde{S}^{-1}$ and $\widetilde{u}$

$$\left(S + uv^\top\right)^{-1} = S^{-1} - \frac{S^{-1}uv^\top S^{-1}}{1 + v^\top S^{-1}u} \tag{34}$$

$$\downarrow \tag{35}$$

$$\left(\widetilde{S} + \widetilde{u}v^\top\right)^{-1} = \widetilde{S}^{-1} - \frac{\widetilde{S}^{-1}\widetilde{u}v^\top \widetilde{S}^{-1}}{1 + v^\top \widetilde{S}^{-1}\widetilde{u}} \tag{36}$$

$$\Updownarrow \tag{37}$$

$$\left(P^\top S + P^\top uv^\top\right)^{-1} = S^{-1}P - \frac{S^{-1}PP^\top uv^\top S^{-1}P}{1 + v^\top S^{-1}PP^\top u} \tag{38}$$

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

$$\widetilde{S}^{-1}\widetilde{S} = S^{-1}PP^\top S \tag{39}$$

$$= S^{-1}IS \tag{40}$$

$$= S^{-1}S \tag{41}$$

B Woodbury with zero-padding

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

$$(S+UV)^{-1} = S^{-1} - S^{-1}U(I+VS^{-1}U)^{-1}VS^{-1} \tag{42}$$

$$\downarrow \tag{43}$$

$$\left(\widetilde{S}+\widetilde{U}V\right)^{-1} = \widetilde{S}^{-1} - \widetilde{S}^{-1}\widetilde{U}(I+V\widetilde{S}^{-1}\widetilde{U})^{-1}V\widetilde{S}^{-1} \tag{44}$$

$$\Updownarrow \tag{45}$$

$$\left(P^\top S+P^\top UV\right)^{-1} = S^{-1}P - S^{-1}PP^\top U(I+VS^{-1}PP^\top U)^{-1}VS^{-1}P \tag{46}$$

References

[1] P. Maponi. The solution of linear systems by using the shermanmorrison formula. *Linear Algebra and its Applications*, 420(2):276–294, 2007.

[2] J. T. Slagel". "the sherman morrison iteration". Master's thesis, "Virginia Polytechnic Institute and State University", "2015".