



HAL
open science

An efficient implementation of Slater-Condon rules

Anthony Scemama, Emmanuel Giner

► **To cite this version:**

Anthony Scemama, Emmanuel Giner. An efficient implementation of Slater-Condon rules. [Research Report] UMR 5626, Laboratoire de Chimie et Physique Quantiques. 2013. hal-01539072

HAL Id: hal-01539072

<https://hal.science/hal-01539072>

Submitted on 29 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An efficient implementation of Slater-Condon rules

Anthony Scemama*, Emmanuel Giner

November 26, 2013

Abstract

Slater-Condon rules are at the heart of any quantum chemistry method as they allow to simplify $3N$ -dimensional integrals as sums of 3- or 6-dimensional integrals. In this paper, we propose an efficient implementation of those rules in order to identify very rapidly which integrals are involved in a matrix element expressed in the determinant basis set. This implementation takes advantage of the bit manipulation instructions on x86 architectures that were introduced in 2008 with the SSE4.2 instruction set. Finding which spin-orbitals are involved in the calculation of a matrix element doesn't depend on the number of electrons of the system.

In this work we consider wave functions Ψ expressed as linear combinations of Slater determinants D of orthonormal spin-orbitals $\phi(\mathbf{r})$:

$$\Psi = \sum_i c_i D_i \quad (1)$$

Using the Slater-Condon rules,[1, 2] the matrix elements of any one-body (\mathcal{O}_1) or two-body (\mathcal{O}_2) operator expressed in the determinant space have simple expressions involving one- and two-electron integrals in the spin-orbital space. The diagonal elements are given by:

$$\begin{aligned} \langle D | \mathcal{O}_1 | D \rangle &= \sum_{i \in D} \langle \phi_i | \mathcal{O}_1 | \phi_i \rangle \\ \langle D | \mathcal{O}_2 | D \rangle &= \frac{1}{2} \sum_{(i,j) \in D} \langle \phi_i \phi_j | \mathcal{O}_2 | \phi_i \phi_j \rangle - \\ &\quad \langle \phi_i \phi_j | \mathcal{O}_2 | \phi_j \phi_i \rangle \end{aligned} \quad (2)$$

For two determinants which differ only by the substitution of spin-orbital i with spin-orbital j :

$$\begin{aligned} \langle D | \mathcal{O}_1 | D_i^j \rangle &= \langle \phi_i | \mathcal{O}_1 | \phi_j \rangle \\ \langle D | \mathcal{O}_2 | D_i^j \rangle &= \sum_{k \in D} \langle \phi_i \phi_k | \mathcal{O}_2 | \phi_j \phi_k \rangle - \langle \phi_i \phi_k | \mathcal{O}_2 | \phi_k \phi_j \rangle \end{aligned} \quad (3)$$

For two determinants which differ by two spin-orbitals:

$$\begin{aligned} \langle D | \mathcal{O}_1 | D_{ik}^{jl} \rangle &= 0 \\ \langle D | \mathcal{O}_2 | D_{ik}^{jl} \rangle &= \langle \phi_i \phi_k | \mathcal{O}_2 | \phi_j \phi_l \rangle - \langle \phi_i \phi_k | \mathcal{O}_2 | \phi_l \phi_j \rangle \end{aligned} \quad (4)$$

All other matrix elements involving determinants with more than two substitutions are zero.

An efficient implementation of those rules requires:

1. to find the number of spin-orbital substitutions between two determinants
2. to find which spin-orbitals are involved in the substitution
3. to compute the phase factor if a reordering of the spin-orbitals has occurred

This paper proposes an efficient implementation of those three points by using some specific bit manipulation instructions at the CPU level.

1 Algorithm

In this section, we use the convention that the least significant bit of binary integers is the rightmost bit. As the position number of a bit in an integer is the exponent for the corresponding bit weight in base 2, the bit positions are numbered from the right to the

*Laboratoire de Chimie et Physique Quantiques, CNRS-IRSAMC, Université de Toulouse, France.

Algorithm 1: Compute the degree of excitation between D_1 and D_2 .

Function `n_excitations`(I^1, I^2)

Input: I^1, I^2 : lists of integers representing determinants D_1 and D_2 .

Output: `d`: Degree of excitation.

```

1 two_d ← 0;
2 for σ ∈ {α, β} do
3   for i ← 0 to Nint - 1 do
4     two_d ← two_d + popcnt (Ii,σ1 xor Ii,σ2);
5 d ← two_d/2;
6 return d;
```

left starting at position 0. To be consistent with this convention, we also represent the arrays of 64-bit integers from right to left, starting at position zero. Following with the usual notations, the spin-orbitals start with index one.

1.1 Binary representation of the determinants

The molecular spin-orbitals in the determinants are ordered by spin: the α spin-orbitals are placed before the β spin-orbitals. Each determinant is represented as a pair of bit-strings: one bit-string corresponding to the α spin-orbital occupations, and one bit-string for the β spin-orbital occupations. When the i -th orbital is occupied by an electron with spin σ in the determinant, the bit at position $(i - 1)$ of the σ bit-string is set to one, otherwise it is set to zero.

The pair of bit-strings is encoded in a 2-dimensional array of 64-bit integers. The first dimension contains N_{int} elements and starts at position zero. N_{int} is the minimum number of 64-bit integers needed to encode the bit-strings:

$$N_{\text{int}} = \lfloor N_{\text{MOs}}/64 \rfloor + 1 \quad (5)$$

where N_{MOs} is the total number of molecular spin-orbitals with spin α or β (we assume this number to be the same for both spins). The second index of the array corresponds to the α or β spin. Hence, determinant D_k is represented by an array of N_{int} 64-bit integers $I_{i,\sigma}^k$, $i \in [0, N_{\text{int}} - 1]$, $\sigma \in \{1, 2\}$.

1.2 Finding the number of substitutions

We propose an algorithm to count the number of substitutions between two determinants D_1 and D_2 (algorithm 1). This number is equivalent to the degree of excitation d of the operator \hat{T}_d which transforms D_1 into D_2 ($D_2 = \hat{T}_d D_1$). The degree of excitation is equivalent to the number of holes created in D_1 or to the number of particles created in D_2 . In algorithm 1, the total number of substitutions is calculated as the sum of the number of holes created in each 64-bit integer of D_1 .

On line 4, $(I_{i,\sigma}^1 \text{ xor } I_{i,\sigma}^2)$ returns a 64-bit integer with bits set to one where the bits differ between $I_{i,\sigma}^1$ and $I_{i,\sigma}^2$. Those correspond to the positions of the holes and the particles. The `popcnt` function returns the number of non-zero bits in the resulting integer. At line 5, `two_d` contains the sum of the number of holes and particles, so the excitation degree `d` is half of `two_d`.

The Hamming weight is defined as the number of non-zero bits in a binary integer. The fast calculation of Hamming weights is crucial in various domains of computer science such as error-correcting codes[3] or cryptography[4]. Therefore, the computation of Hamming weights has appeared in the hardware of processors in 2008 via the `popcnt` instruction introduced with the SSE 4.2 instruction set. This instruction has a 3-cycle latency and a 1-cycle throughput independently of the number of bits set to one (here, independently of the number of electrons), as opposed to Wegner’s algorithm[5] that repeatedly finds and clears the last nonzero bit. The `popcnt` instruction may be generated by Fortran compilers via the intrinsic `popcnt` function.

1.3 Identifying the substituted spin-orbitals

Algorithm 2 creates the list of spin-orbital indices containing the holes of the excitation from D_1 to D_2 . At line 4, H is set to a 64-bit integer with ones at the positions of the holes. The loop starting at line 5 translates the positions of those bits to spin-orbital indices as follows: when $H \neq 0$, the index of the rightmost bit of H set to one is equal to the number of trailing zeros of the integer. This number can be obtained by the x86_64 `bsf` (bit scan forward) instruction with a latency of 3 cycles and a 1-cycle throughput, and may be generated by the Fortran `trailz` intrinsic function. At line 7, the spin-orbital

Algorithm 2: Obtain the list of orbital indices corresponding to holes in the excitation from D_1 to D_2

Function `get_holes`(I^1, I^2);

Input: I^1, I^2 : lists of integers representing determinants D_1 and D_2 .

Output: Holes: List of positions of the holes.

```

1 for  $\sigma \in \{\alpha, \beta\}$  do
2   k  $\leftarrow$  0;
3   for i  $\leftarrow$  0 to  $N_{\text{int}} - 1$  do
4     H  $\leftarrow$  ( $I_{i,\sigma}^1$  xor  $I_{i,\sigma}^2$ ) and  $I_{i,\sigma}^1$ ;
5     while H  $\neq$  0 do
6       position  $\leftarrow$  trailing_zeros(H);
7       Holes[k,  $\sigma$ ]  $\leftarrow$  1 + 64  $\times$  i + position;
8       H  $\leftarrow$  bit_clear(H, position);
9       k  $\leftarrow$  k + 1;
10 return Holes;
```

index is calculated. At line 8, the rightmost bit set to one is cleared in H .

The list of particles is obtained in a similar way with algorithm 3.

1.4 Computing the phase

In our representation, the spin-orbitals are always ordered by increasing index. Therefore, a reordering may occur during the spin-orbital substitution, involving a possible change of the phase.

As no more than two substitutions between determinants D_1 and D_2 give a non-zero matrix element, we only consider in algorithm 4 single and double substitutions. The phase is calculated as $-1^{N_{\text{perm}}}$, where N_{perm} is the number permutations necessary to bring the spin-orbitals on which the holes are made to the positions of the particles. This number is equal to the number of occupied spin-orbitals between these two positions.

We create a bit mask to extract the occupied spin-orbitals placed between the hole and the particle, and we count them using the `popcnt` instruction. We have to consider that the hole and the particle may or may not belong to the same 64-bit integer.

On lines 6 and 7, we identify the highest and lowest spin-orbitals involved in the excitation to delimitate the range of the bit mask. Then, we find to which 64-bit integers they belong and what are their bit positions in the integers (lines 8–11). The loop in lines 12–13 sets to one all the bits of the mask con-

Algorithm 3: Obtain the list of orbital indices corresponding to particles in the excitation from D_1 to D_2

Function `get_particles`(I^1, I^2)

Input: I^1, I^2 : lists of integers representing determinants D_1 and D_2 .

Output: Particles: List of positions of the particles.

```

1 for  $\sigma \in \{\alpha, \beta\}$  do
2   k  $\leftarrow$  0;
3   for i  $\leftarrow$  0 to  $N_{\text{int}} - 1$  do
4     P  $\leftarrow$  ( $I_{i,\sigma}^1$  xor  $I_{i,\sigma}^2$ ) and  $I_{i,\sigma}^2$ ;
5     while P  $\neq$  0 do
6       position  $\leftarrow$  trailing_zeros(P);
7       Particles[k,  $\sigma$ ]  $\leftarrow$  1 + 64  $\times$  i + position;
8       P  $\leftarrow$  bit_clear(P, position);
9       k  $\leftarrow$  k + 1;
10 return Particles;
```

tained in the integers between the integer containing the lowest orbital (included) and the integer containing highest orbital (excluded). Line 14 sets all the m rightmost bits of the integer to one and all the other bits to zero. At line 15, the $n + 1$ rightmost bits of the integer containing to the lowest orbital are set to zero. At this point, the bit mask is defined on integers `mask[j]` to `mask[k]` (if the substitution occurs on the same 64-bit integer, $j = k$). `mask[j]` has zeros on the leftmost bits and `mask[k]` has zeros on the rightmost bits. We can now extract the spin-orbitals placed between the hole and the particle by applying the mask and computing the Hamming weight of the result (line 17).

For a double excitation, if the realization of the first excitation introduces a new orbital between the hole and the particle of the second excitation (crossing of the two excitations), an additional permutation is needed, as done on lines 18–19. This *if* statement assumes that the Holes and Particles arrays are sorted.

2 Optimized Implementation

In this section, we present our implementation in the Fortran language. In Fortran, arrays start by default with index one as opposed to the convention chosen in the algorithms.

A Fortran implementation of algorithm 1 is given in Figure 1. This function was compiled with the Intel

Algorithm 4: Compute the phase factor of $\langle D_1 | \mathcal{O} | D_2 \rangle$

Function GetPhase(Holes, Particles)

Input: Holes and Particles obtained with algorithms 2 and 3.

Output: phase $\in \{-1, 1\}$.

```

1 Requires: n_excitations( $I^1, I^2$ )  $\in \{1, 2\}$ .
   Holes and Particles are sorted.
2 nperm  $\leftarrow 0$ ;
3 for  $\sigma \in \{\alpha, \beta\}$  do
4    $n_\sigma \leftarrow$  Number of excitations of spin  $\sigma$ ;
5   for  $i \leftarrow 0$  to  $n_\sigma - 1$  do
6     high  $\leftarrow$  max(Particles[ $i, \sigma$ ], Holes[ $i, \sigma$ ]);
7     low  $\leftarrow$  min(Particles[ $i, \sigma$ ], Holes[ $i, \sigma$ ]);
8     k  $\leftarrow$   $\lfloor$ high/64 $\rfloor$ ;
9     m  $\leftarrow$  high (mod 64);
10    j  $\leftarrow$   $\lfloor$ low/64 $\rfloor$ ;
11    n  $\leftarrow$  low (mod 64);
12    for l  $\leftarrow j$  to k - 1 do
13      mask[l]  $\leftarrow$  not(0);
14    mask[k]  $\leftarrow 2^m - 1$ ;
15    mask[j]  $\leftarrow$  mask[j] and (not( $2^{n+1}$ )+1)
16    for l  $\leftarrow j$  to k do
17      nperm  $\leftarrow$ 
18      | nperm+popcnt( $I_1^{j,\sigma}$  and mask[l]);
19  if ( $n_\sigma = 2$ ) and ( Holes[2,  $\sigma$ ] < Particles[1,  $\sigma$ ]
   or Holes[1,  $\sigma$ ] > Particles[2,  $\sigma$ ] ) then
20    | nperm  $\leftarrow$  nperm + 1;
21 return  $-1^{nperm}$ ;

```

Fortran compiler 14.0.0 with the options `-xAVX -O2`. A static analysis of the executable was performed using the MAQAO tool[6] : if all the data fit into the L1 cache, an iteration of the loop takes in average 3.5 CPU cycles on an Intel Sandy Bridge CPU core. A dynamic analysis revealed 10.5 cycles for calling the function, performing the first statement and exiting the function, and 4.1 CPU cycles per loop iteration. Therefore, to obtain the best performance this function will need to be inlined by the compiler, eventually using compiler directives or inter-procedural optimization flags.

For the identification of the substitutions, only four cases are possible (figure 2) depending of the degree of excitation : no substitution, one substitution, two substitutions or more than two substitutions. If the determinants are the same, the subroutine exits with a degree of excitation of zero. For degrees of exci-

```

1 integer function n_excitations(det1,det2,Nint)
2 implicit none
3 integer*8, intent(in) :: det1(Nint,2), det2(Nint,2)
4 integer , intent(in) :: Nint
5
6 integer :: l
7
8 n_excitations = &
9   popcnt(xor( det1(1,1), det2(1,1)) ) + &
10  popcnt(xor( det1(1,2), det2(1,2)) )
11
12 do l=2,Nint
13   n_excitations = n_excitations + &
14   popcnt(xor( det1(l,1), det2(l,1)) ) + &
15   popcnt(xor( det1(l,2), det2(l,2)) )
16 end do
17 n_excitations = ishft(n_excitations,-1)
18
19 end

```

Figure 1: Fortran implementation of algorithm 1.

tation higher than two, the subroutine returns a degrees of excitation equal to -1. For the two remaining cases, a particular subroutine is written for each case. The cases were ordered from the most probable to the least probable to optimize the branch prediction. In output, the indices of the spin-orbitals are given in the array `exc` as follows:

- The last index of `exc` is the spin (1 for α and 2 for β)
- The second index of `exc` is 1 for holes and 2 for particles
- The element at index 0 of the first dimension of `exc` gives the total number of holes or particles of spin α or β
- The first index of `exc` refers to the orbital index of the hole or particle

The subroutine for single excitations is given in figure 3. The particle and hole are searched simultaneously. The `ishift` variable (line 19) allows to replace the integer multiplication at line 7 of algorithm 2 by an integer addition, which is faster. Line 38 contains a bit shift instruction where all the bits are shifted 6 places to the right. This is equivalent to doing the integer division of `high` by 64. Line 39 computes `high (mod 64)` using a bit mask. The compiler may recognize that those last two optimizations are possible, but it can not be aware that `high` is always positive. Therefore, it will generate additional instructions to handle negative integers. As we know that `high` is always positive, we can do better than the compiler. The test at line 35 is true when both the hole and the particle have been found. This allows to compute the phase factor and exit the subroutine as soon as the

```

1 subroutine get_excitation(det1,det2,exc,degree,phase,Nint)
2 implicit none
3 integer, intent(in) :: Nint
4 integer*8, intent(in) :: det1(Nint,2), det2(Nint,2)
5 integer, intent(out) :: exc(0:2,2,2)
6 integer, intent(out) :: degree
7 double precision, intent(out) :: phase
8
9 integer :: n_excitations
10
11 degree = n_excitations(det1,det2,Nint)
12
13 select case (degree)
14
15   case (3:)
16     degree = -1
17     return
18
19   case (2)
20     call get_double_excitation(det1,det2,exc,phase,Nint)
21     return
22
23   case (1)
24     call get_single_excitation(det1,det2,exc,phase,Nint)
25     return
26
27   case(0)
28     return
29
30   end select
31 end

```

Figure 2: Fortran subroutine for finding the holes and particles involved in a matrix element.

single excitation is obtained. If the hole and particle belong to the same integer (line 42), the bit mask is created and applied on the fly to only one 64-bit integer. Otherwise, the bit mask is created and applied on the two extreme integers j and k . The integers between j and k , if there are any, don't need to have a bit mask applied since the mask would have all bits set to one. Finally, line 52 calculates $-1^{N_{\text{perm}}}$ using a memory access depending on the parity of N_{perm} : `phase_dble` is an array of two double precision values (line 10).

For double excitations, the subroutine is similar to the previous one. The `nexc` variable counts how many holes and particles have been found, in order to exit the loop (line 45) as soon as the double excitation is found. The calculation of the phase is the same as the case of single excitations (lines 48–67), but in the case of a double excitation of the same spin, orbital crossings can occur (lines 68–76).

3 Benchmarks

All the benchmarks were realized on a quad-core Intel Xeon CPU E3-1220 @ 3.10GHz (8 MiB cache). The benchmark program is single-threaded, and was run using a single CPU core at the maximum turbo frequency of 3.4 GHz. The numbers of CPU cycles

```

1 subroutine get_single_excitation(det1,det2,exc,phase,Nint)
2 implicit none
3 integer, intent(in) :: Nint
4 integer*8, intent(in) :: det1(Nint,2)
5 integer*8, intent(in) :: det2(Nint,2)
6 integer, intent(out) :: exc(0:2,2,2)
7 double precision, intent(out) :: phase
8 integer :: tz, l, ispin, ishft, nperm, i, j, k, m, n, high, low
9 integer*8 :: hole, particle, tmp
10 double precision, parameter :: phase_dble(0:1) = (/ 1.d0, -1.d0 /)
11
12 exc(0,1,1) = 0
13 exc(0,2,1) = 0
14 exc(0,1,2) = 0
15 exc(0,2,2) = 0
16 do ispin = 1,2
17   ishft = -63
18   do l=1,Nint
19     ishft = ishft + 64
20     if (det1(l,ispin) == det2(l,ispin)) cycle
21     tmp = xor( det1(l,ispin), det2(l,ispin) )
22     particle = iand(tmp, det2(l,ispin))
23     hole = iand(tmp, det1(l,ispin))
24     if (particle /= 0_8) then
25       tz = trailz(particle)
26       exc(0,2,ispin) = 1
27       exc(1,2,ispin) = tz+ishft
28     end if
29     if (hole /= 0_8) then
30       tz = trailz(hole)
31       exc(0,1,ispin) = 1
32       exc(1,1,ispin) = tz+ishft
33     end if
34
35     if ( iand(exc(0,1,ispin),exc(0,2,ispin)) == 1 ) then
36       low = min(exc(1,1,ispin),exc(1,2,ispin))
37       high = max(exc(1,1,ispin),exc(1,2,ispin))
38       j = ishft(low-1,-6)+1
39       n = iand(low,63)
40       k = ishft(high-1,-6)+1
41       m = iand(high,63)
42       if (j=k) then
43         nperm = popcnt(iand(det1(j,ispin), &
44           iand( ibset(0_8,m-1)-1_8, ibclr(-1_8,n)+1_8 ) ))
45       else
46         nperm = popcnt(iand(det1(k,ispin), ibset(0_8,m-1)-1_8)) + &
47           popcnt(iand(det1(j,ispin), ibclr(-1_8,n) +1_8))
48         do i=j+1,k-1
49           nperm = nperm + popcnt(det1(i,ispin))
50         end do
51       end if
52       phase = phase_dble(iand(nperm,1))
53       return
54     end if
55   end do
56 end do
57 end

```

Figure 3: Fortran subroutine for finding the holes and particles involved in a single excitation.

```

1 subroutine get_double_excitation(det1,det2,exc,phase,Nint)
2 implicit none
3 integer, intent(in) :: Nint
4 integer*8, intent(in) :: det1(Nint,2), det2(Nint,2)
5 integer, intent(out) :: exc(0:2,2,2)
6 double precision, intent(out) :: phase
7 integer :: l, ispin, idx_hole, idx_particle, ishift
8 integer :: i,j,k,m,n,high, low,a,b,c,d,nperm,tz,nexc
9 integer*8 :: hole, particle, tmp
10 double precision, parameter :: phase_dble(0:1) = (/ 1.d0, -1.d0 /)
11 exc(0,1,1) = 0
12 exc(0,2,1) = 0
13 exc(0,1,2) = 0
14 exc(0,2,2) = 0
15 nexc=0
16 nperm=0
17 do ispin = 1,2
18   idx_particle = 0
19   idx_hole = 0
20   ishift = -63
21   do l=1,Nint
22     ishift = ishift + 64
23     if (det1(l,ispin) == det2(l,ispin)) then
24       cycle
25     end if
26     tmp = xor( det1(l,ispin), det2(l,ispin) )
27     particle = iand(tmp, det2(l,ispin))
28     hole = iand(tmp, det1(l,ispin))
29     do while (particle /= 0_8)
30       tz = trailz(particle)
31       nexc = nexc+1
32       idx_particle = idx_particle + 1
33       exc(0,2,ispin) = exc(0,2,ispin) + 1
34       exc(idx_particle,2,ispin) = tz+ishift
35       particle = iand(particle,particle-1_8)
36     end do
37     do while (hole /= 0_8)
38       tz = trailz(hole)
39       nexc = nexc+1
40       idx_hole = idx_hole + 1
41       exc(0,1,ispin) = exc(0,1,ispin) + 1
42       exc(idx_hole,1,ispin) = tz+ishift
43       hole = iand(hole,hole-1_8)
44     end do
45     if (nexc == 4) exit
46   end do
47
48   do i=1,exc(0,1,ispin)
49     low = min(exc(i,1,ispin),exc(i,2,ispin))
50     high = max(exc(i,1,ispin),exc(i,2,ispin))
51     j = ishft(low-1,-6)+1
52     n = iand(low,63)
53     k = ishft(high-1,-6)+1
54     m = iand(high,63)
55     if (j==k) then
56       nperm = nperm + popcnt(iand(det1(j,ispin), &
57         iand( ibset(0_8,m-1)-1_8, ibclr(-1_8,n)+1_8 ) ))
58     else
59       nperm = nperm + popcnt(iand(det1(k,ispin), &
60         ibset(0_8,m-1)-1_8)) &
61         + popcnt(iand(det1(j,ispin), &
62         ibclr(-1_8,n) +1_8))
63       do l=j+1,k-1
64         nperm = nperm + popcnt(det1(l,ispin))
65       end do
66     end if
67   end do
68   if (exc(0,1,ispin) == 2) then
69     a = min(exc(1,1,ispin), exc(1,2,ispin))
70     b = max(exc(1,1,ispin), exc(1,2,ispin))
71     c = min(exc(2,1,ispin), exc(2,2,ispin))
72     d = max(exc(2,1,ispin), exc(2,2,ispin))
73     if (c>a .and. c<b .and. d>b) nperm = nperm + 1
74   exit
75   end if
76 end do
77 phase = phase_dble(iand(nperm,1))
78
79 end

```

Figure 4: Fortran subroutine for finding the holes and particles involved in a double excitation.

were obtained by polling the hardware time stamp counter `rdtscp` using the following C function:

```

double rdtscp_(void) {
  unsigned long long a, d;
  __asm__ volatile ("rdtscp" : "=a" (a),
                    "=d" (d));
  return (double)((d<<32) + a);
}

```

Two systems were benchmarked. Both systems are a set of 10 000 determinants obtained with the CIPSI algorithm presented in ref [7]. The first system is a water molecule in the cc-pVTZ basis set[8], in which the determinants are made of 5 α - and 5 β -electrons in 105 molecular orbitals ($N_{\text{int}} = 2$). The second system is a Copper atom in the cc-pVDZ[9] basis set, in which the determinants are made of 15 α - and 14 β -electrons in 49 molecular orbitals ($N_{\text{int}} = 1$). The benchmark consists in comparing each determinant with all the derminants (10^8 determinant comparisons). These determinant comparisons are central in determinant driven calculations, such as the calculation of the Hamiltonian matrix in the determinant basis set. As an example of a practical application, we benchmark the calculation of the one-electron density matrix on the molecular orbital basis using the subroutine given in figure 5.

The programs were compiled with the Intel Fortran Compiler version 14.0.0. The compiling options included inter-procedural optimization to let the compiler inline functions via the `-ipo` option, and the instruction sets were specified using the `-xAVX` or the `-xSSE2` options. Let us recall that the AVX instruction set includes all the instructions introduced with SSE4.2. Using AVX instructions, the `popcnt` function is executed through its hardware implementation, as opposed to SSE2 in which `popcnt` is executed through its software implementation.

In table 1 we report measures of the CPU time and of the average number of CPU cycles of all the subroutines presented in this paper for the water molecule and the Copper atom.

The measures of `n_excitations` and `get_excitation` are made using all the possible pairs of determinants. Most of the time the degree of excitation d is greater than 2, so the average number of cycles has a large weight on the $d > 2$ case, and this explains why the average number of cycles is much lower than in the $d = 1$ and $d = 2$ cases.

As the computation of the one-electron density matrix only requires to find the spin-orbitals for the

```

1 subroutine compute_density_matrix(det,Ndet,coef,mo_num, &
2   Nint,density_matrix)
3 implicit none
4 integer*8, intent(in)      :: det(Nint,2,Ndet)
5 integer, intent(in)       :: Ndet, Nint, mo_num
6 double precision, intent(in) :: coef(Ndet)
7 double precision, intent(out) :: density_matrix(mo_num,mo_num)
8
9 integer :: i,j,k,l,ispin,ishift
10 integer*8 :: buffer
11 integer :: deg
12 integer :: exc(0:2,2,2)
13 double precision :: phase, c
14 integer :: n_excitations
15
16 density_matrix = 0.d0
17 do k=1,Ndet
18   do ispin=1,2
19     ishift = 0
20     do i=1,Nint
21       buffer = det(i,ispin,k)
22       do while (buffer /= 0_8)
23         j = trailz(buffer) + ishift
24         density_matrix(j,j) = density_matrix(j,j) &
25           + coef(k)*coef(k)
26         buffer = iand(buffer,buffer-1_8)
27       end do
28       ishift = ishift+64
29     end do
30   end do
31   do l=1,k-1
32     if (n_excitations(det(1,1,k),det(1,1,l),Nint) /= 1) then
33       cycle
34     end if
35     call get_excitation(det(1,1,k),det(1,1,l),exc,deg,phase,Nint)
36     if (exc(0,1,1) == 1) then
37       i = exc(1,1,1)
38       j = exc(1,2,1)
39     else
40       i = exc(1,1,2)
41       j = exc(1,2,2)
42     end if
43     c = phase*coef(k)*coef(l)
44     c = c+c
45     density_matrix(j,i) = density_matrix(j,i) + c
46     density_matrix(i,j) = density_matrix(i,j) + c
47   end do
48 end do
49 end

```

Figure 5: Fortran subroutine for the calculation of the one-electron density matrix in the molecular orbital basis.

cases $d \in \{0,1\}$, which are a very small fraction of the total, the average number of cycles is very close to this of `n_excitations`. Note that the calculation of the density matrix only requires $N(N+1)/2$ determinant comparisons, and this explains why the CPU time is smaller than for the `n_excitations` benchmark.

All the source files needed to reproduce the benchmark presented in this section are available at https://github.com/scemama/slater_condon.

4 Summary

We have presented an efficient implementation of Slater-Condon rules by taking advantage of instructions recently introduced in x86_64 processors. The use of these instructions allow to gain a factor larger than 6 with respect to their software implementation.

	Time (s)		Cycles	
	AVX	SSE2	AVX	SSE2
H₂O				
<code>n_excitations</code>	0.33	2.33	10.2	72.6
<code>get_excitation</code>	0.60	2.63	18.4	81.4
<code>get_excitation, d = 0</code>			6.2	58.7
<code>get_excitation, d = 1</code>			53.0	126.3
<code>get_excitation, d = 2</code>			88.9	195.5
<code>get_excitation, d > 2</code>			6.7	63.6
Density matrix	0.19	1.23	11.7	75.7
Cu				
<code>n_excitations</code>	0.17	1.13	5.3	35.2
<code>get_excitation</code>	0.28	1.27	8.7	39.1
<code>get_excitation, d = 0</code>			4.9	30.4
<code>get_excitation, d = 1</code>			47.0	88.1
<code>get_excitation, d = 2</code>			78.8	145.5
<code>get_excitation, d > 2</code>			5.5	29.3
Density matrix	0.10	0.63	6.5	38.9

Table 1: CPU time (seconds) and average number of CPU cycles measured for the `n_excitations` function, the `get_excitation` subroutine and the calculation of the one-electron density matrix in the molecular orbital basis. `get_excitation` was also called using selected pairs of determinants such that the degree of excitation (d) was zero, one, two or higher.

As a result, the computation of the degree of excitation between two determinants can be performed in the order of 10 CPU cycles in a set of 128 molecular orbitals, independently of the number of electrons. Obtaining the list of holes and particles involved in a single or double excitation can be obtained in the order of 50–90 cycles, also independently of the number of electrons. For comparison, the latency of a double precision floating point division is typically 20–25 cycles, and a random read in memory is 250–300 CPU cycles. Therefore, the presented implementation of Slater-Condon rules will significantly accelerate determinant-driven calculations where the two-electron integrals have to be fetched using random memory accesses. As a practical example, the one-electron density matrix built from 10 000 determinants of a water molecule in the cc-pVTZ basis set was computed in 0.2 seconds on a single CPU core.

References

- [1] J. C. Slater. The theory of complex spectra. *Phys. Rev.*, 34:1293–1322, Nov 1929.

- [2] E. U. Condon. The theory of complex spectra. *Phys. Rev.*, 36:1121–1133, Oct 1930.
- [3] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.
- [4] Y. Hilewitz, Z.J. Shi, and R.B. Lee. Comparing fast implementations of bit permutation instructions. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, volume 2, pages 1856–1863 Vol.2, 2004.
- [5] Peter Wegner. A technique for counting ones in a binary computer. *Commun. ACM*, 3(5):322–, May 1960.
- [6] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. MAQAO: Modular assembler quality Analyzer and Optimizer for Itanium 2. In *Workshop on EPIC Architectures and Compiler Technology, San Jose, California, United-States*, March 2005.
- [7] Emmanuel Giner, Anthony Scemama, and Michel Caffarel. Using perturbatively selected configuration interaction in quantum monte carlo calculations. *Canadian Journal of Chemistry*, 91(9):879–885, September 2013.
- [8] Thom H. Dunning. Gaussian basis sets for use in correlated molecular calculations. i. the atoms boron through neon and hydrogen. *The Journal of Chemical Physics*, 90(2):1007–1023, 1989.
- [9] Nikolai B. Balabanov and Kirk A. Peterson. Systematically convergent basis sets for transition metals. i. all-electron correlation consistent basis sets for the 3d elements sczn. *The Journal of Chemical Physics*, 123(6):064107, 2005.