Single core optimization in QMC=Chem

Anthony Scemama <scemama@irsamc.ups-tlse.fr>
Michel Caffarel <michel.caffarel@irsamc.ups-tlse.fr>

Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

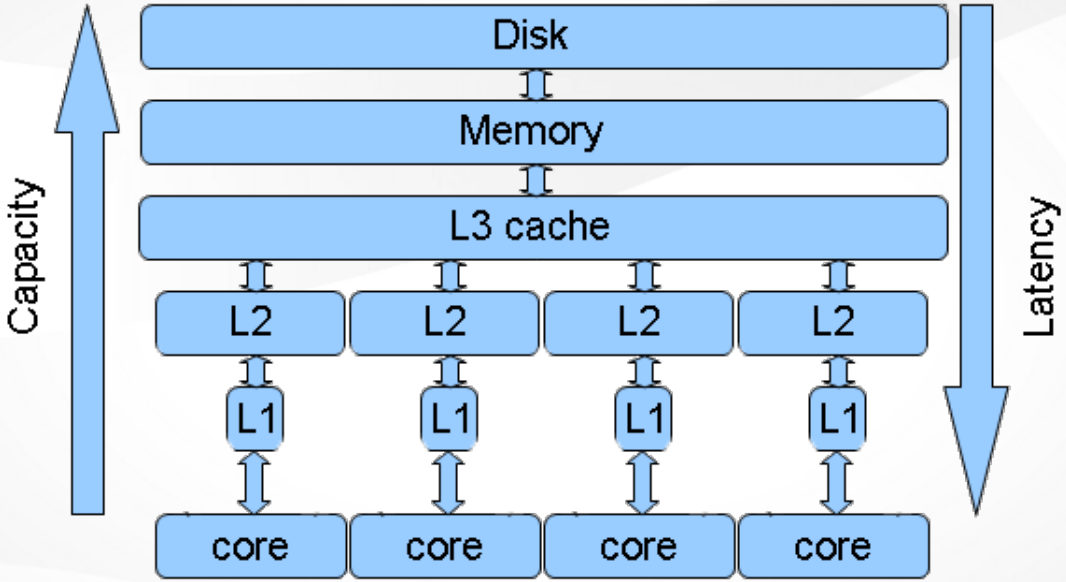
Hardware considerations

Intel(R) Xeon(R) CPU E31220 @ 3.10GHz
3.4GHz turbo, 8 MiB shared L3, 256 KiB L2, 32 KiB L1

- The AVX instruction set allows to perform vector operations on 256 bits : 8 single precision (SP) elements or 4 double precision (DP) elements
- The vector ADD and MUL operations have a throughput of 1 per cycle (pipelining)
- One vector ADD, one vector MUL and one integer ADD (loop count) can be performed simultaneously
- Therefore, the peak performance of an Intel Sandy/Ivy Bridge core is 16 floating point operations (flops) per cycle (SP) or 8 flops/cycle (DP)
- One E31220 core has a peak performance of 54.4 Gflops/s (SP), 27.2 Gflops/s (DP)
- In the peak regime, one flop takes in average 0.018 ns (SP) or 0.037 ns (DP)

On modern architectures, reducing the number of flops does **not** systematically reduce the execution time.

Memory access is much more critical. Understanding how the data arrives to the CPU helps to write efficient code [*].



---

* "**What Every Programmer Should Know About Memory**, U. Drepper, (2007), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.957

Measures obtained with LMbench †

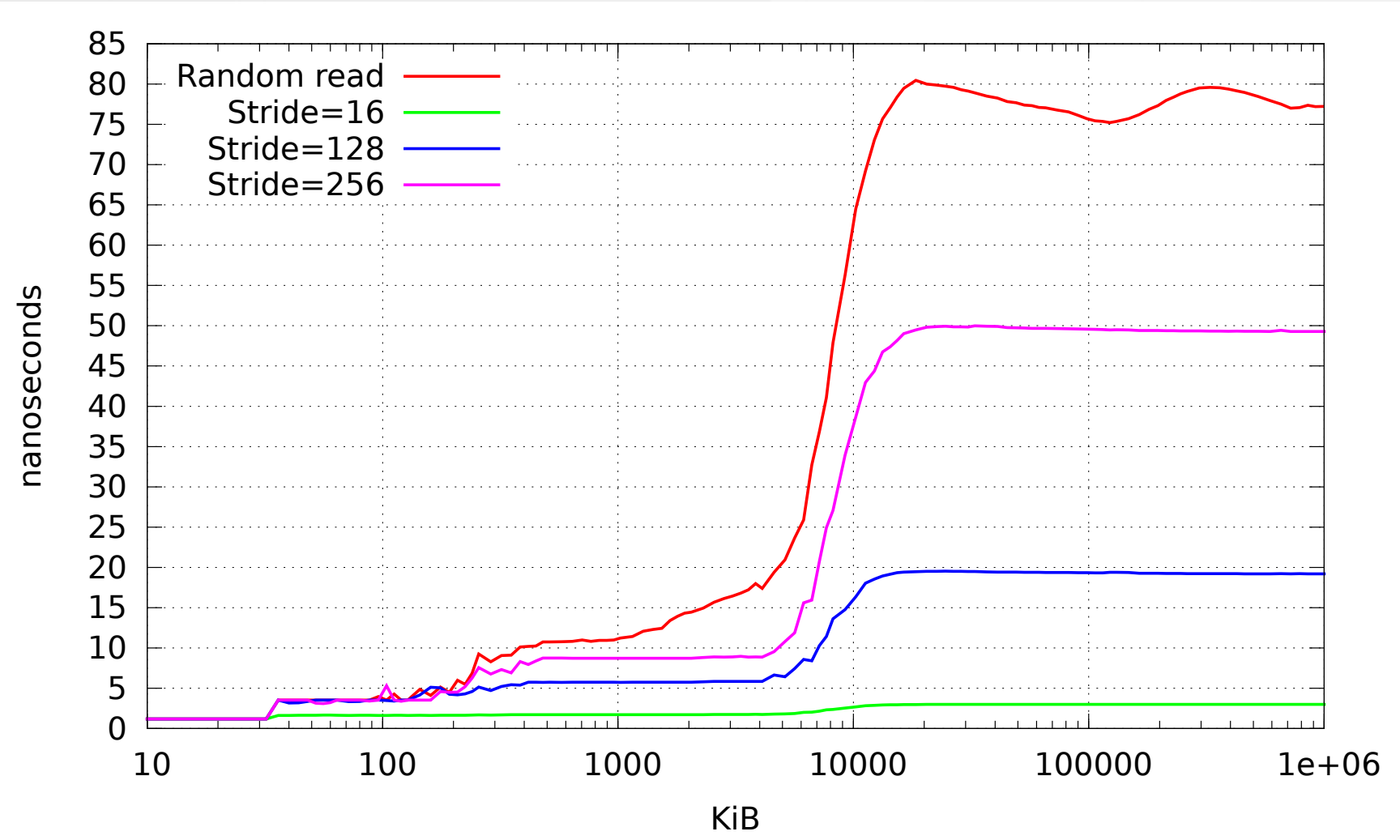
1 cycle = 0.29 ns, 1 peak flop SP = 0.018 ns

Integer (ns)	bit	ADD	MUL	DIV	MOD
32 bit	0.3	0.04	0.9	6.7	7.7
64 bit	0.3	0.04	0.9	13.2	12.9

Floating Point (ns)	ADD	MUL	DIV
32 bit	0.9	1.5	4.4
64 bit	0.9	1.5	6.8

Data read (ns)	Random	Prefetched
L1	1.18	1.18
L2	3.5	1.6
L3	13	1.7
Memory	75-80	3.

† http://www.bitmover.com/lmbench/


Random read
Stride=16
Stride=128
Stride=256
nanoseconds
KiB
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
10
100
1000
10000
100000
1e+06

- Random access to memory is **very** slow : 79 ns = 270 CPU cycles : 4300 flops (Peak SP)
- Strided access to memory with a stride < 4096 KiB (1 page) triggers the hardware prefetchers, reducing the memory latencies. Smaller strides are better, and give latencies comparable to L2 latencies.

Recomputing data may be faster than fetching it randomly in memory

Other important numbers:

Mutex lock/unlock	~100 ns
Infiniband	~1 200 ns
Ethernet	~50 000 ns
Disk seek (SSD)	~50 000 ns
Disk seek (15k rpm)	~2 000 000 ns

Example : squared distance matrix

```
do j=1,n
 do i=1,j
  dist1(i,j) = X(i,1)*X(j,1) + X(i,2)*X(j,2) + X(i,3)*X(j,3)
 end do
end do

do j=1,n
 do i=j+1,n
  dist1(i,j) = dist1(j,i)
 end do
end do
```

t(n=133) = 13.0 μs, 3.0 GFlops/s
t(n=4125) = 95.4 ms, 0.44 GFlops/s

```
do j=1,n
 do i=1,j
  dist2(i,j) = X(i,1)*X(j,1) + X(i,2)*X(j,2) + X(i,3)*X(j,3)
  dist2(j,i) = dist2(i,j)
 end do
end do
```

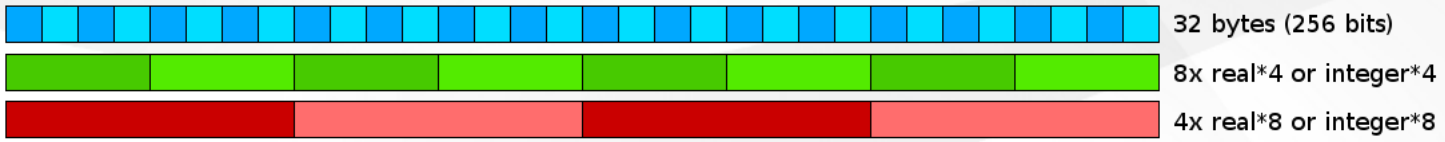
t(n=133) = 11.5 μs : 1.13x speed-up, 3.5 GFlops/s
t(n=4125) = 90.4 ms : 1.05x speed-up, 0.47 GFlops/s

```
do j=1,n
 do i=1,n
  dist3(i,j) = X(i,1)*X(j,1) + X(i,2)*X(j,2) + X(i,3)*X(j,3)
 end do
end do
```

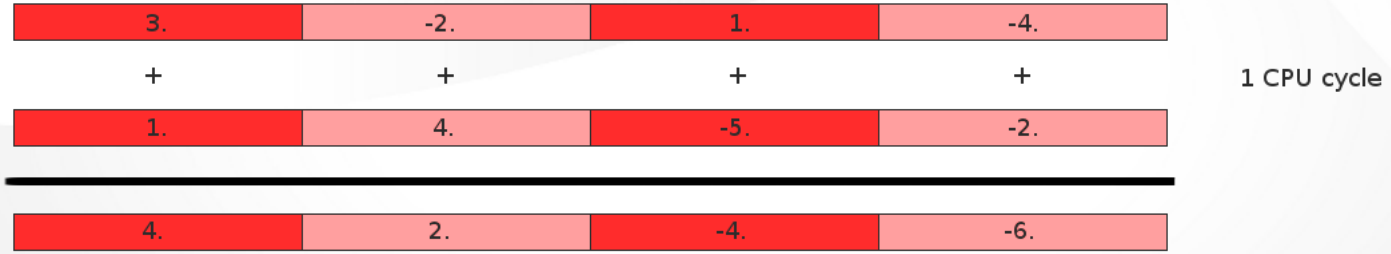
2x more flops!
t(n=133) = 10.3 μs : 1.12x speed up, 8.2 GFlops/s
t(n=4125) = 15.7 ms : 5.75x speed up, 5.4 GFlops/s

Vector operations

AVX single instruction / multiple data (SIMD) instructions operate on 256-bit floating point registers:



Example : vector ADD in double precision:



Requirements:

- The elements of each SIMD vector must be contiguous in memory
- The first element of each SIMD vector must be aligned on a 32 byte boundary

Automatic vectorization

The compiler can generate automatically vector instructions when possible. An auto-vectorized loop generates 3 loops:

Peel loop (scalar)

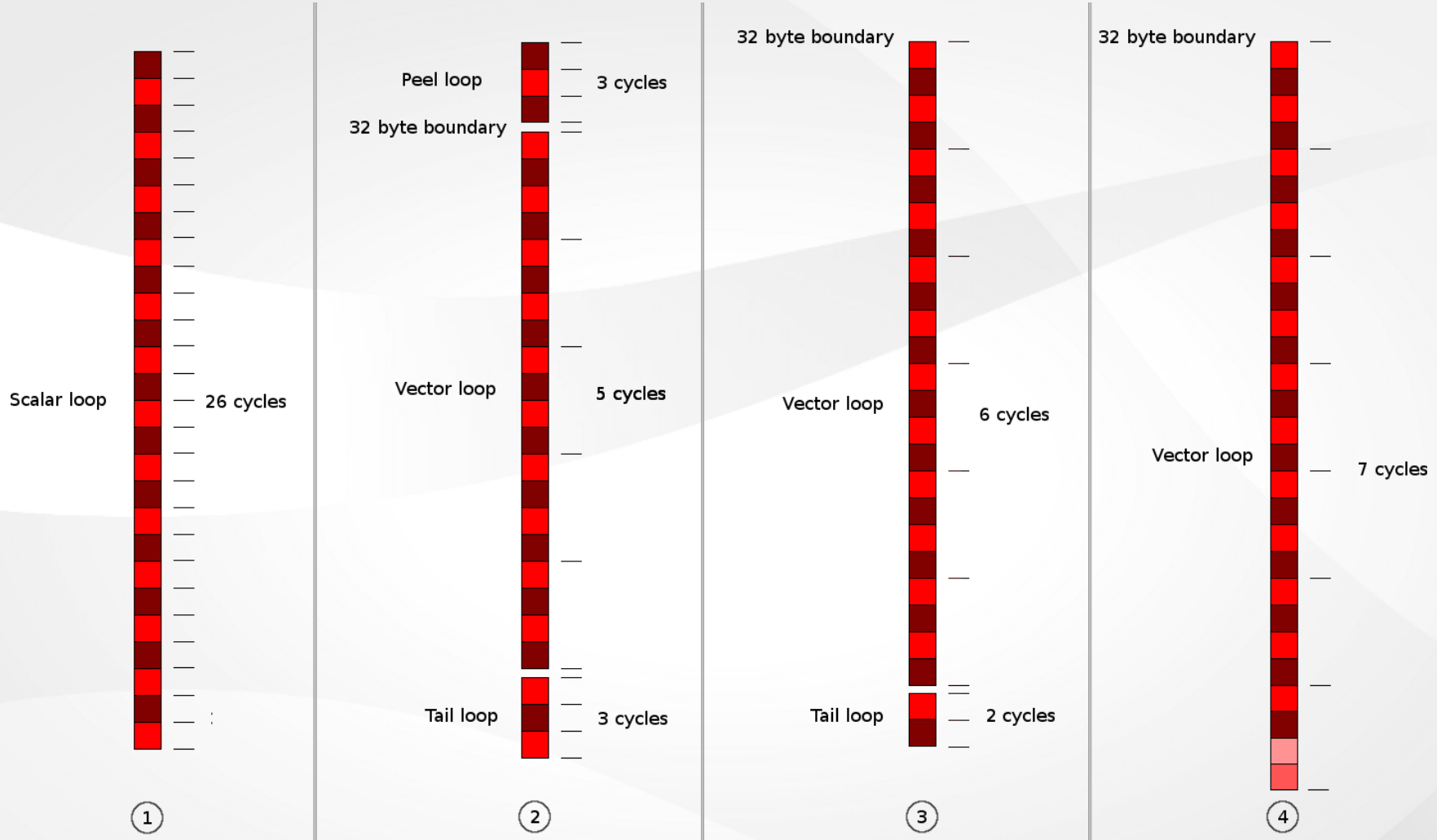
First elements until the 32 byte boundary is met

Vector loop

Vectorized version until the last vector of 4 elements

Tail loop (scalar)

Last elements


Scalar loop
26 cycles
1
Peel loop
3 cycles
32 byte boundary
Vector loop
5 cycles
Tail loop
3 cycles
2
32 byte boundary
Vector loop
6 cycles
Tail loop
2 cycles
3
32 byte boundary
Vector loop
7 cycles
4

Intel specific Compiler directives

To remove the peel loop, you can tell the compiler to align the arrays on a 32 byte boundary using:

```
double precision, allocatable :: A(:), B(:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A, B
```

Then, before using the arrays in a loop, you can tell the compiler that the arrays are aligned. Be careful: if one array is not aligned, this may cause a segmentation fault.

```
!DIR$ VECTOR ALIGNED
do i=1,n
  A(i) = A(i) + B(i)
end do
```

To remove the tail loop, you can allocate A such that its dimension is a multiple of 4 elements:

```
n_4 = mod(n,4)
if (n_4 == 0) then
  n_4 = n
else
  n_4 = n - n_4 + 4
endif
allocate ( A(n_4), B(n_4) )
```

and rewrite the loop as follows:

```
do i=1,n,4
  !DIR$ VECTOR ALIGNED
  !DIR$ VECTOR ALWAYS
  do k=0,3
    A(i+k) = A(i+k) + B(i+k)
  end do
end do
```

In that case, the compiler knows that each inner-most loop cycle can be transformed safely into only vector instructions, and it will not produce the tail and peel loops with the branching. For small arrays, the gain can be significant.

For multi-dimensional arrays, if the 1st dimension is a multiple of 4 elements, all the columns are aligned:

```
double precision, allocatable :: A(:,:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A
allocate( A(n_4,m) )
do j=1,m
 do i=1,n,4
   !DIR$ VECTOR ALIGNED
   !DIR$ VECTOR ALWAYS
   do k=0,3
     A(i+k,j) = A(i+k,j) * B(i+k,j)
   end do
 end do
end do
```

Warning :

In practice, using multiples of 4 elements is not always the best choice. Using multiples of 8 or 16 elements can be better because the inner-most loop may be unrolled by the compiler to improve the efficiency of the pipeline.

Example : squared distance matrix

```
do j=1,n
 do i=1,n,8
  !DIR$ VECTOR ALIGNED
  !DIR$ VECTOR ALWAYS
  do k=0,7
    dist4(i+k,j) = X(i+k,1)*X(j,1) + X(i+k,2)*X(j,2) + X(i+k,3)*X(j,3)
  end do
 end do
end do
```

t(n=133) = 7.2 μs : 1.44x speed-up, 12.1 GFlops/s
t(n=4125) = 15.5 ms : 1.01x speed-up, 7.5 GFlops/s.

Hot spots of QMC algorithms

At every Monte Carlo step, the following quantities have to be computed:

- $\Psi_T \; : \; \Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N) = \sum_i c_i \det\left(D_i^\alpha(\mathbf{r}_1, \ldots, \mathbf{r}_{N_\alpha})\right) \det\left(D_i^\beta(\mathbf{r}_{N_\alpha+1}, \ldots, \mathbf{r}_N)\right)$

  Slater matrix:

  $$D_i^\alpha(\mathbf{r}_1, \ldots, \mathbf{r}_{N_\alpha}) = \begin{pmatrix} \phi_1(\mathbf{r}_1) & \ldots & \phi_{N_\alpha}(\mathbf{r}_1) \\ \vdots & \vdots & \vdots \\ \phi_1(\mathbf{r}_{N_\alpha}) & \ldots & \phi_{N_\alpha}(\mathbf{r}_{N_\alpha}) \end{pmatrix}$$

- $\dfrac{\nabla_i \Psi_T}{\Psi_T} \; : \; \dfrac{1}{\Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N)} \dfrac{\partial}{\partial \mathbf{r}_i} \Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N)$

- $\dfrac{\Delta_i \Psi_T}{\Psi_T} \; : \; \dfrac{1}{\Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N)} \Delta_i \Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N)$

Calculation of the Slater matrices

The Slater matrices have to be computed, as well as their gradients and Laplacian.

It is necessary to compute the values, gradients and Laplacian of the Molecular Orbitals (MOs) at the electron positions.

$$\phi_i(\mathrm{r}) = \sum_k C_{ik}\chi_k(\mathrm{r})$$

$$\chi_k(\mathrm{r}) = (x - x_A)^{a_k}(y - y_A)^{b_k}(z - z_A)^{c_k}\sum_l d_l e^{-\alpha_{kl}|\mathrm{r}-\mathrm{r}_A|^2}$$

- *C* is the matrix of MO coefficients (constant)
- A_1 : MO values
- B_1 : AO values
- A_2, A_3, A_4 : MO gradients (x,y,z)

- B_2, B_3, B_4 : AO gradients (x,y,z)
- A_5 : of MO Laplacian
- B_5 : of AO Laplacian

We need to compute $A_i = C \times B_i$ efficiently:

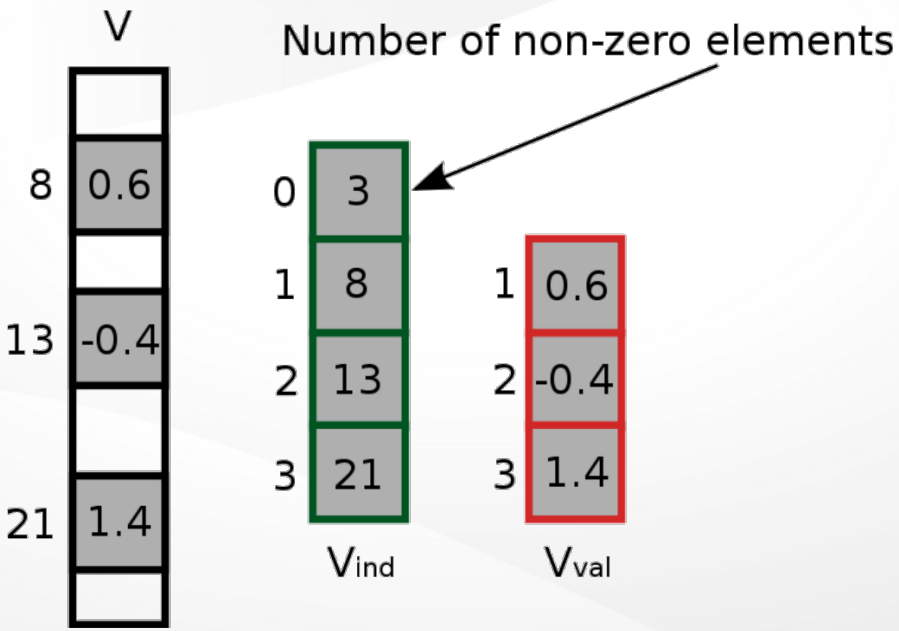
- Single precision is sufficient
- AOs are not orthonormal and centered on nuclei
- All B_i have null elements where $|r-r_i|^2$ is large : only non-zero elements are computed
- All B_i are sparse with non-zero elements at the same indices
- C is constant and dense
- The size of A_i is small (~ $N_{elec} / 2$)
- We have implemented a very efficient dense x sparse matrix product for small matrices

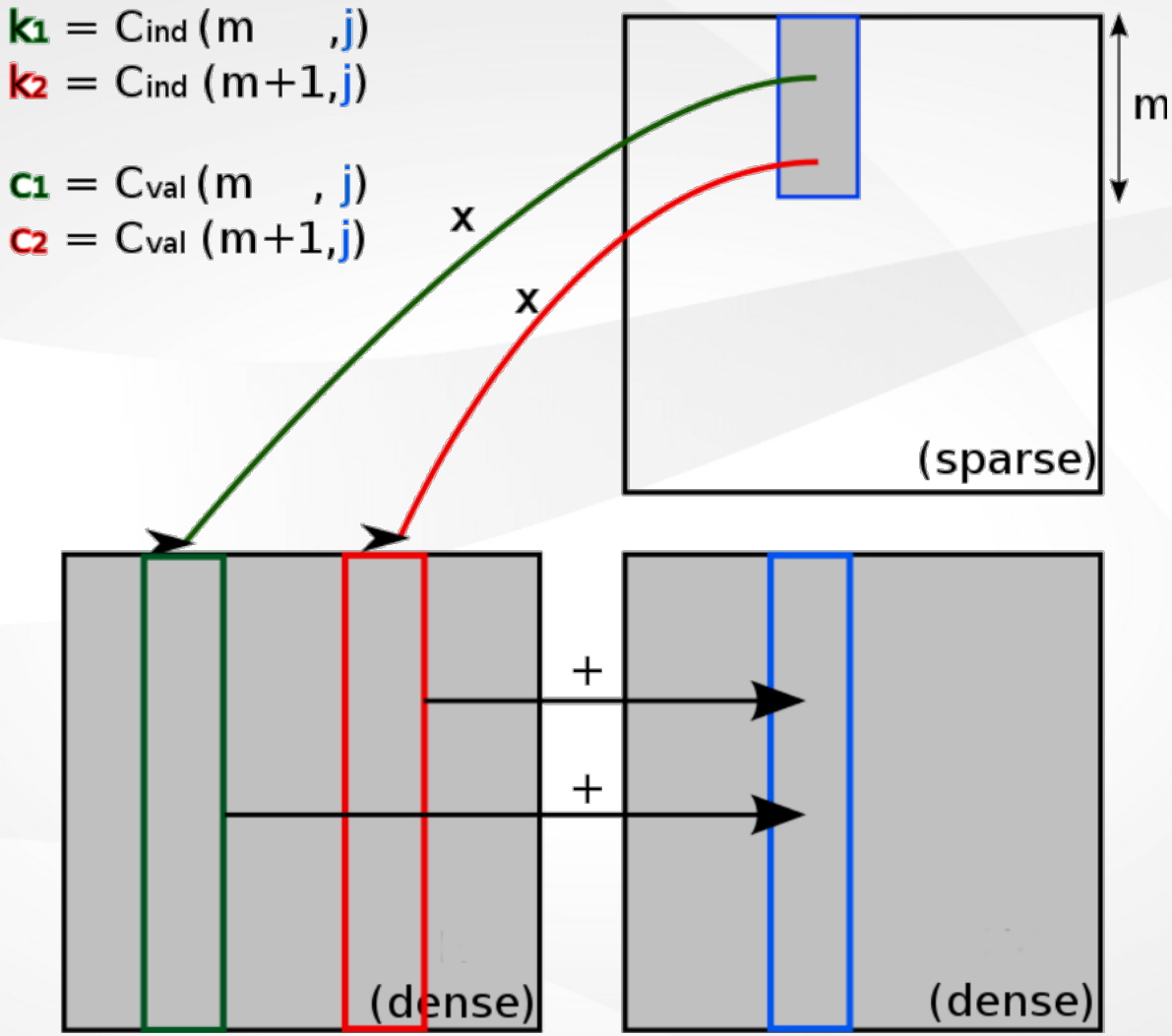
Dense Matrix x Sparse Vector Product

To improve cache locality and reduce memory, we:

- compute one column of all B_i and store them sparse
- make the product of C with these vectors and store all A_i

The sparse vectors are represented as:




k1 = Cind (m ,j)
k2 = Cind (m+1,j)
c1 = Cval (m , j)
c2 = Cval (m+1,j)
x
x
m
(sparse)
+
+
(dense)
(dense)

In QMC=Chem, all arrays are aligned on a 32 byte boundary by IRPF90. The leading dimension is always a multiple of 8 elements.

```
! $IRP_ALIGN = 32
! $IRP_ALIGN/4-1 = 7

! Initialize output vectors
! -------------------------

     !DIR$ VECTOR ALIGNED
     do j=1,LDA,max(1,$IRP_ALIGN/4)
      !DIR$ VECTOR ALIGNED
      A1(j:j+$IRP_ALIGN/4-1) = 0.
      !DIR$ VECTOR ALIGNED
      A2(j:j+$IRP_ALIGN/4-1) = 0.
      !DIR$ VECTOR ALIGNED
      A3(j:j+$IRP_ALIGN/4-1) = 0.
      !DIR$ VECTOR ALIGNED
      A4(j:j+$IRP_ALIGN/4-1) = 0.
```

```
      !DIR$ VECTOR ALIGNED
       A5(j:j+$IRP_ALIGN/4-1) = 0.
     enddo

! Unroll and jam x 4
! -------------------

     kmax2 = indices(0)-mod(indices(0),4)
     do kao=1,kmax2,4

  ! Fetch column indices
  ! --------------------

      k_vec(1) = indices(kao  )
      k_vec(2) = indices(kao+1)
      k_vec(3) = indices(kao+2)
      k_vec(4) = indices(kao+3)
```

```
! Fetch column factors (1,2)
! ---------------------------

    d11 = B1(kao  )
    d21 = B1(kao+1)
    d31 = B1(kao+2)
    d41 = B1(kao+3)

    d12 = B2(kao  )
    d22 = B2(kao+1)
    d32 = B2(kao+2)
    d42 = B2(kao+3)

! A += C x B (1,2)
! ----------------

    !DIR$ VECTOR ALIGNED
    !DIR$ LOOP COUNT (256)
```

```
      do j=1,LDA,max(1,$IRP_ALIGN/4)

        !DIR$ VECTOR ALIGNED
        A1(j:j+$IRP_ALIGN/4-1) = A1(j:j+$IRP_ALIGN/4-1) + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d11 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d21 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d31 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d41

        !DIR$ VECTOR ALIGNED
        A2(j:j+$IRP_ALIGN/4-1) = A2(j:j+$IRP_ALIGN/4-1) + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d12 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d22 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d32 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d42

      enddo
```

```
! Fetch column factors (3,4)
! ---------------------------

    d13 = B3(kao  )
    d23 = B3(kao+1)
    d33 = B3(kao+2)
    d43 = B3(kao+3)

    d14 = B4(kao  )
    d24 = B4(kao+1)
    d34 = B4(kao+2)
    d44 = B4(kao+3)

! A = C x B (3,4)
! ---------------

    !DIR$ VECTOR ALIGNED
    do j=1,LDA,max(1,$IRP_ALIGN/4)
```

```
        !DIR$ VECTOR ALIGNED
        A3(j:j+$IRP_ALIGN/4-1) = A3(j:j+$IRP_ALIGN/4-1) + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d13 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d23 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d33 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d43

        !DIR$ VECTOR ALIGNED
        A4(j:j+$IRP_ALIGN/4-1) = A4(j:j+$IRP_ALIGN/4-1) + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d14 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d24 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d34 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d44

      enddo

  ! Fetch column factors (5)
```

```
! ---------------------------

    d15 = B5(kao  )
    d25 = B5(kao+1)
    d35 = B5(kao+2)
    d45 = B5(kao+3)


! A += C x B (5), unrolled 2x by compiler
! ---------------------------------------

    !DIR$ VECTOR ALIGNED
    do j=1,LDA,max(1,$IRP_ALIGN/4)  ! Unroll 2 times

      !DIR$ VECTOR ALIGNED
      A5(j:j+$IRP_ALIGN/4-1) = A5(j:j+$IRP_ALIGN/4-1) + &
         C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d15 + &
         C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d25 + &
```

```
            C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d35 + &
            C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d45

       enddo

     enddo

! Tail loop of outer loop
! ------------------------

     do kao = kmax2+1, indices(0)

  ! Fetch column indice
  ! -------------------

       k_vec(1) = indices(kao)
```

```
! Fetch column factors (1-5)
! ---------------------------

      d11 = B1(kao)
      d12 = B2(kao)
      d13 = B3(kao)
      d14 = B4(kao)
      d15 = B5(kao)


! A += B x C (1-5)
! ----------------

      !DIR$ VECTOR ALIGNED
      do j=1,LDA,max(1,$IRP_ALIGN/4)

       !DIR$ VECTOR ALIGNED
       A1(j:j+$IRP_ALIGN/4-1) = A1(j:j+$IRP_ALIGN/4-1) + &
```

```
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d11

    !DIR$ VECTOR ALIGNED
    A2(j:j+$IRP_ALIGN/4-1) = A2(j:j+$IRP_ALIGN/4-1) + &
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d12

    !DIR$ VECTOR ALIGNED
    A3(j:j+$IRP_ALIGN/4-1) = A3(j:j+$IRP_ALIGN/4-1) + &
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d13

    !DIR$ VECTOR ALIGNED
    A4(j:j+$IRP_ALIGN/4-1) = A4(j:j+$IRP_ALIGN/4-1) + &
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d14

    !DIR$ VECTOR ALIGNED
    A5(j:j+$IRP_ALIGN/4-1) = A5(j:j+$IRP_ALIGN/4-1) + &
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d15
```

```
        enddo
      enddo
```

Inner-most loops:

- Perfect ADD/MUL balance
- Does not saturate load/store units
- Only vector operations with no peel/tail loops
- Uses 15 AVX registers. No register spilling
- If all data fits in L1, 100% peak is reached (16 flops/cycle)
- In practice: memory bound, so 50-60% peak is measured.

Other QMC codes use 3D splines to avoid the computation of AOs, and the matrix products but:

- To do the 3D interpolation, 8 values are needed (corners of a cube)
- This represents 4 random memory accesses : ~320 nanoseconds + the time to compute the interpolation
- In 360 nanoseconds, we can do ~ 12 000 flops.
- The average computation time of 1 element with our matrix product is proportional to the number of non-zero elements in B_i (<500 flops).
- We have shown that our implementation (calculation of A_i and matrix products) is faster than the interpolation by factors of 1.0x to 1.5x
- 3D splines have to be pre-computed on a grid. It takes initialization time
- 3D splines needs many GiB of RAM, so only small systems can be handled, and OpenMP parallelism is often required.

Inverse Slater matrices

To compute $\frac{\nabla_i \Psi_T}{\Psi_T}$ and $\frac{\Delta_i \Psi_T}{\Psi_T}$, one needs the inverse of the Slater matrices:

- $\frac{\nabla_i \Psi_T}{\Psi_T}$ needs $D_k^{-1} \nabla_i D_k$
- $\frac{\Delta_i \Psi_T}{\Psi_T}$ needs $D_k^{-1} \Delta_i D_k$

A unique list of alpha and beta Slater matrices is generated, and the results are combined at the end to produce the alpha x beta determinant products.

To give accurate results, double precision is required. For each spin, the first inverse Slater matrix is fully calculated:

- < 5x5 : hand-written O(N!) algorithm ($5! < 5^3$)
- > 5x5 : MKL library : *dgetrf*, *dgetri*

The next determinants are calculated using the Shermann-Morisson-Woodbury formula : if only one column differs, the new inverse can be computed in $O(N^2)$.

A CAS-SCF wave function with 10 000 determinant products has 100 unique alpha and 100 unique beta determinants. One of those will be computed in $O(N^3)$ and all others will be computed in $O(N^2)$. For a 40 electrons system (20 alpha, 20 beta), computing 10 000 determinants will be only ~6x longer than the single-determinant calculation.