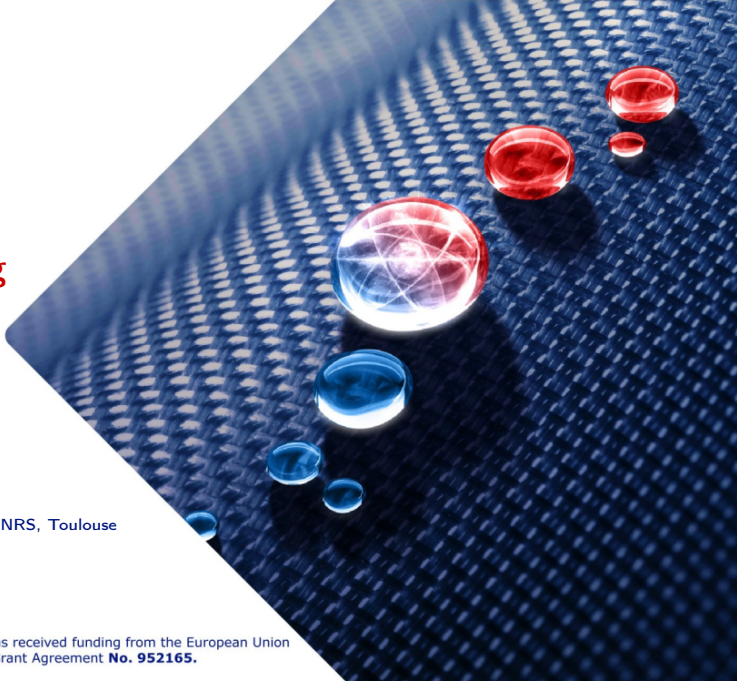
Targeting Real chemical accuracy at the EXascale

Guidelines for improving the performance of computer programs

Anthony Scemama

16/04/2021

Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse (France)



Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizoon 2020 research and innovation programme under Grant Agreement **No. 952165.**

TREX: Targeting Real chemical accuracy at the EXascale

- European Center of Excellence for HPC
- A dozen of groups (HPC scientists, Supercomputing centers, quantum chemists/pysicists)
- Goal: bring high accuracy methods on future exascale systems
  - Quantum Monte Carlo (QMC)
  - Selected CI (CIPSI, FCIQMC)
  - Multireference SAPT (GammCor)
- Software libraries:
  - QMCkl : a library containing all the computational kernels of QMC
  - TREXIO: a library for storing wave functions, for code inter-operability
- `https://trex-coe.eu`

DGEMM: Double precision GEneral Matrix-Matrix multiplication

Important take-home message

- Express everything in terms of dense Matrix Multiplications

Importance of Data movement

Example: Coulomb Matrix

$$J_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma} \iint \chi_\mu(r_1)\chi_\lambda(r_2)\frac{1}{r_{12}}\chi_\nu(r_1)\chi_\sigma(r_2)dr_1 dr_2$$

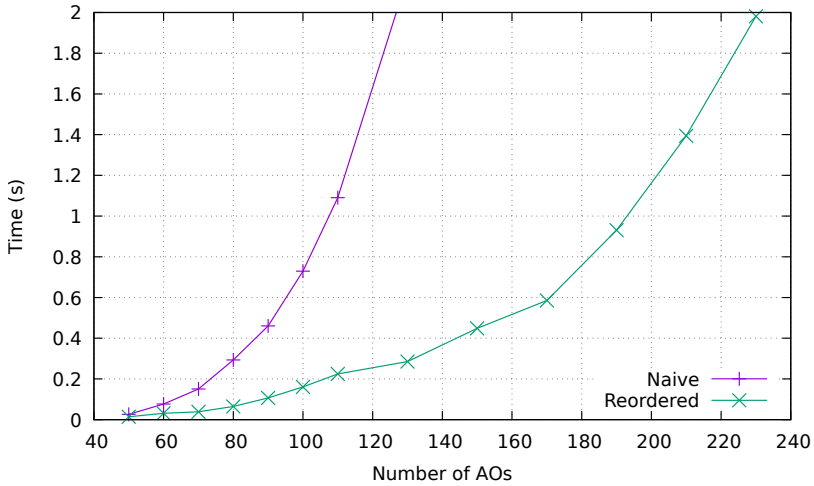
```
1     do mu=1,N
2        do nu=1,N
3           J(mu,nu) = 0.d0
4           do la=1,N
5              do si=1,N
6                 J(mu,nu) = J(mu,nu) + P(la,si) * ERI(mu,la,nu,si)
7              end do
8           end do
9        end do
10    end do
```

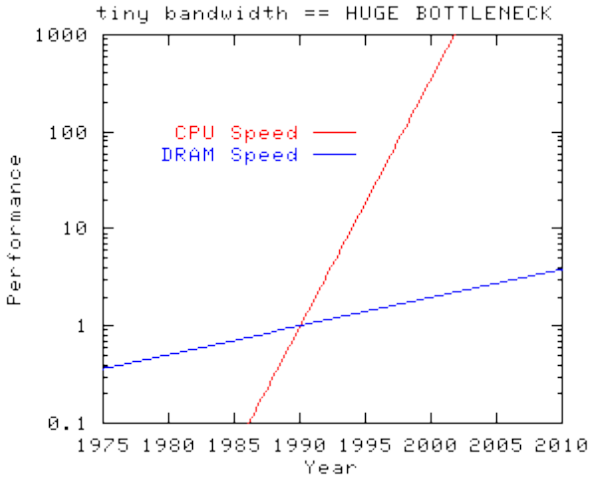
$$J_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma} \iint \chi_\mu(r_1)\chi_\lambda(r_2)\frac{1}{r_{12}}\chi_\nu(r_1)\chi_\sigma(r_2)dr_1 dr_2$$

```
1     do nu=1,N
2        do mu=1,N
3           J(mu,nu) = 0.d0
4           do si=1,N
5              do la=1,N
6                 J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
7              end do            !-----------!
8           end do
9        end do
10    end do
```

We changed the order of indices in the storage of integrals.

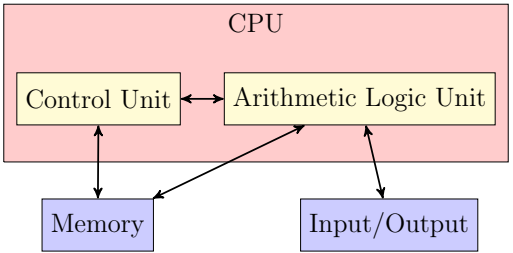
Example: Coulomb Matrix





- CPU : $\times 1.55$ / year
- Memory : $\times 1.10$ / year

Stream benchmark : `http://www.cs.virginia.edu/stream`



- Feed continuously the Control Unit with instructions
- Feed the Arithmetic Logic Units continuously with data

Bandwidth Throughput : Amount of data which goes through one point per unit of time

Latency Time to transfer *one* element between two points

	Latency	Bandwidth
	Low $(300km/h)^{-1}$	Low 2
	High $(80km/h)^{-1}$	High 68

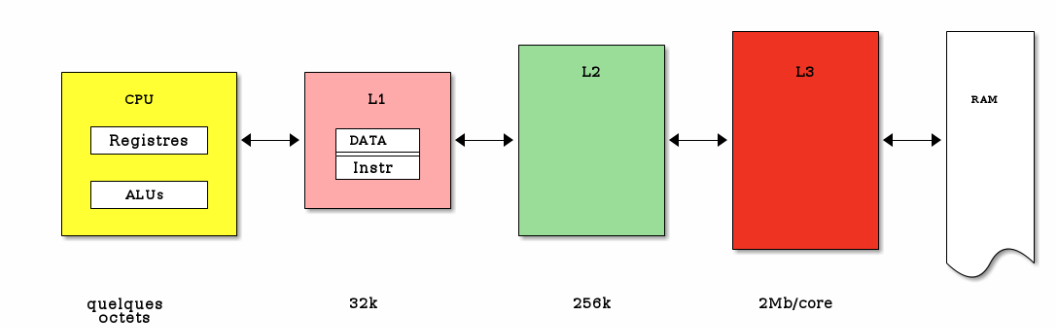
Bandwidth Throughput : Amount of data which goes through one point per unit of time

Latency Time to transfer *one* element between two points

Over 20 years:

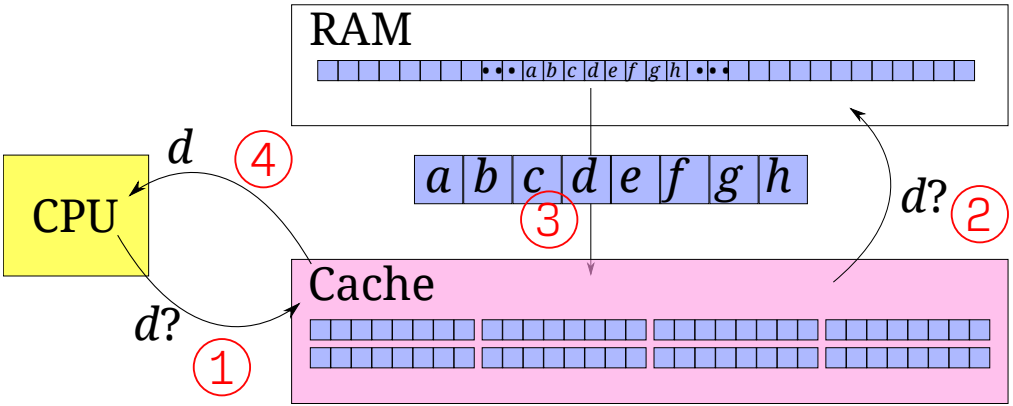
	Latency	Bandwidth
Disk	8×	143×
RAM	4×	120×
Ethernet	16×	1000×
CPU	21×	2250×

- Random access is more and more expensive (latency-bound)
- Hierarchical memories (caches) : hide latencies



- One ALU makes a `LOAD` d
- If $d \in$ L1, copy from d to the register
- If $d \notin$ L1,
- If $d \in$ L2, copy from d to L1 and in the register
- *etc*

When a cache asks for some data at a higher level, it transfers a cache line : a block of fixed size (typically 64 bytes)



Locality

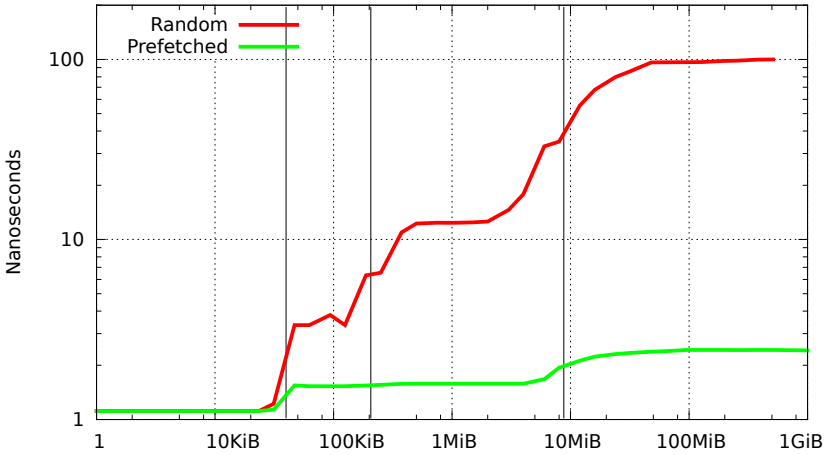
Spatial : If the CPU asks for e after having asked for d, e will be in the cache

Temporal : The cache line replaced by the new one is the *least recently used* (LRU)

Prefetching

If a regular access is detected, the next cache lines will be asked for in advance.

Access in an increasingly large array:



Latency (Nanoseconds)

Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz

1 cycle = 0.28 ns, peak SP throughput (AVX2) = 0.0087 ns/flop

Integer	ADD	MUL	DIV	MOD	Bit
32 bit	0.28	0.84	6.60	7.07	0.28
64 bit	0.28	0.84	11.80	11.75	0.28
Floating Point	ADD	MUL	DIV		
32 bit	0.84	1.39	3.77		
64 bit	0.84	1.39	5.71		
Data read	Random	Prefetched			
L1	1.11	1.11			
L2	3.3	1.54			
L3	12.3	1.58			
RAM	100.	2.4			

`http://lmbench.sourceforge.net`

Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz

1 cycle = 0.28 ns, peak SP throughput (AVX2) = 32 flops/cycle

Integer	ADD	MUL	DIV	MOD	Bit
32 bit	1	3	23	25	1
64 bit	1	3	42	42	1
Floating Point	**ADD**	**MUL**	**DIV**		
32 bit	3	5	13		
64 bit	3	5	20		
Data read	**Random**	**Prefetched**			
L1	4	4			
L2	12	5.5			
L3	44	5.6			
RAM	357	8.6			

`http://lmbench.sourceforge.net`

Latency: Numbers to keep in mind

Operation	Latency (ns)	Scaled
Int ADD	0.3	1 s
FP ADD	0.9	3 s
FP MUL	1.5	5 s
Int32 DIV	6.6	22 s
FP64 DIV	5.7	19 s
L1 cache	1.2	4 s
L2 cache	3.5	12 s
L3 cache	13	43 s
RAM	79	4 min 23 s
Send 4KB with 100 Gbps Infiniband	1 040	57 min 46 s
Send 4KB over 10 Gbps ethernet	10 000	9 h 16 min
Write 4KB randomly to NVMe SSD	30 000	1 day 4 h
Transfer 1MB to/from PCIe GPU	80 000	3 days 11 h
Random Disk Access (seek+rotation)	10 000 000	1 year 21 days

Accessing contiguous data: good!

- Maximizes bandwidth : you get a full cache line per transfer
- Minimizes latency : data is in cache + prefetch

Random access: bad!

- Minimizes bandwidth : you get one useful element per transfer
- Maximizes latency : cache miss + no prefetch

Fortran:
```
allocate (A(n))
```

C:
```
A = (double*) malloc (n*sizeof(double));
```

- `malloc` : Allocates a continuous block of memory
- `sizeof(double)` : number of bytes (8 for a `double`)
- `(double*)` cast from `void*` to `double*` :

enables pointer arithmetic for the computation of the address : (A[i]) $\longrightarrow$ physical (0xa23b4)

The elements of an array are contiguous

Fortran: `allocate (A(n,m))` $\Longrightarrow$ contiguous

C: Multiple possibilities

```
1  A = (double**) malloc(n*sizeof(double*));
2  for (i=1 ; i<n ; i++)
3    A[i] = (double*) malloc(m*sizeof(double) );
```

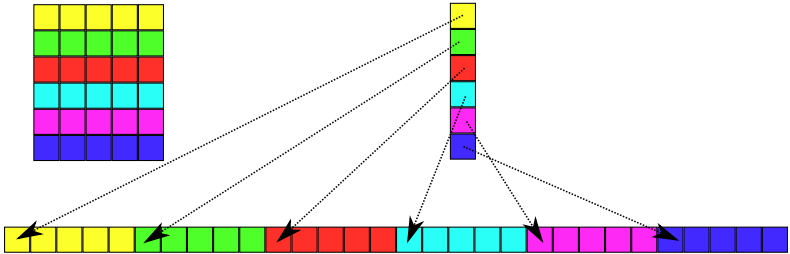
Array of allocated arrays $\Longrightarrow$ *not* contiguous

First possibility : Simulate with a 1D array and a *#define* statement

```
1  A = (double*) malloc(n*m * sizeof(double) );
2
3  #define A(i,j) (A[(i)*m + (j)])
```

Second possibility : build an array of pointers, pointing in a contiguous 1D array

```
1    A    = (double**) malloc( n   * sizeof(double*) );
2    A[0] = (double* ) malloc( n*m * sizeof(double ) );
3    for (i=1 ; i<n ; i++)       // Increment the address
4      A[i] = A[i-1] + m;        // by m*sizeof(double) bytes
```



Row-major and Column-major ordering

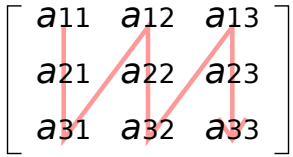
Fortran : Column-major

```
1   do j=1,n
2     do i=1,n ! <-- i : inside j
3       A(i,j)
4     end do
5   end do
```

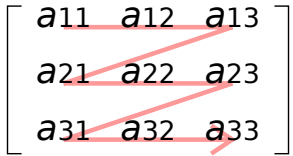
C : Row-major

```
1   for (i=0 ; i<n ; i++) {
2     for (j=0 ; j<n ; j++) { // <-- j inside i
3       A[i][j]
4     }
5   }
```

Column-major



Row-major



Re-ordering array indices

$$J_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma} \int\int \chi_\mu(r_1)\chi_\lambda(r_2)\frac{1}{r_{12}}\chi_\nu(r_1)\chi_\sigma(r_2)dr_1 dr_2$$

```
1  do nu=1,N
2   do mu=1,N
3    J(mu,nu) = 0.d0
4    do si=1,N
5     do la=1,N
6      J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
7     end do
8    end do
9   end do
10 end do
```

Optimizing instructions

When should we optimize instructions?

When we are sure that the code is not memory-bound: computation is much faster than data movement (memory wall).

Arithmetic intensity: $\sigma = N(Flops)/N(bytes)$

- Add two vectors : `x[i] += y[i]`
  $\sigma = N/(24N) = 0.042$ flops/byte memory-bound
- Dot product : $x = \sum_i^N a_i \times b_i$
  $\sigma = 2N/(16N) = 0.125$ flops/byte memory-bound
- Matrix-vector: $X_i = \sum_j^N A_{ij} \times b_j$
  $\sigma = 2N^2/(8N^2 + 16N) \sim 0.25$ flops/byte memory-bound
- Matrix-matrix :$X_{ij} = \sum_k A_{ik} \times B_{kj}$
  $\sigma = 2N^3/(24N^2) \propto N$ CPU-bound

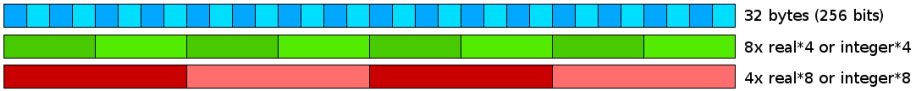
Cheap instructions : high throughput, low latency

- ADD, MUL (int)
- ADD, MUL (float/double)
- AND, OR, XOR, ... (bool)

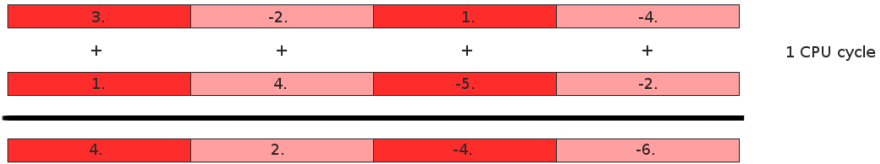
Expensive instructions : low throughput, high latency

- DIV, MOD (int)
- DIV (double)
- SQRT (double)
- EXP, POW, LOG, SIN, COS, etc (float/double)

SIMD : Single Instruction, Multiple Data
Execute the *same* instruction in parallel on all the elements of a vector:



Example : AVX vector ADD in double precision:



Different instruction sets exist in the x86 micro-architecture:

- MMX : Integer (64-bit wide)
- SSE $\rightarrow$ SSE4.2 : Integer and Floating-point (128-bit)
- AVX : Integer and Floating-point (256-bit)
- AVX-512 : Integer and Floating-point (512-bit)

Requirements:

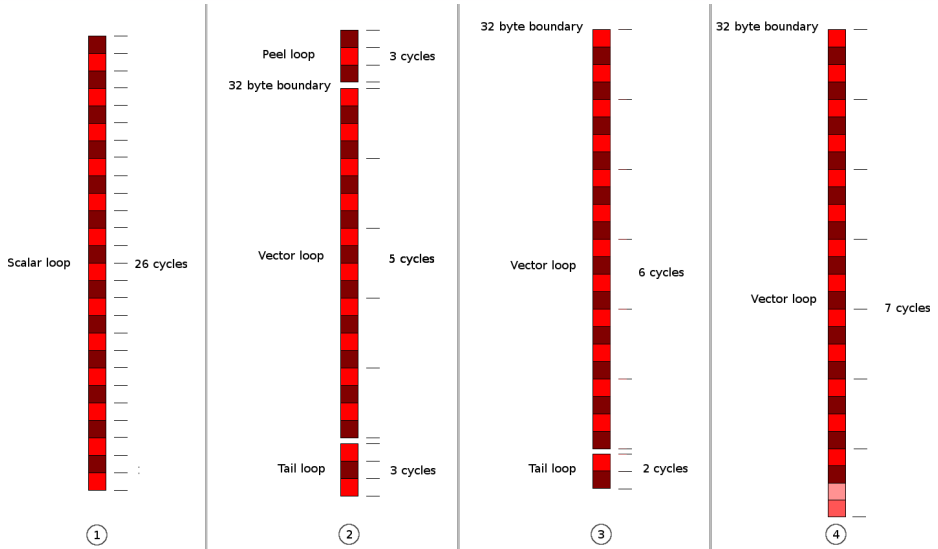
- The elements of each SIMD vector must be contiguous in memory
- The first element of each SIMD vector must be *aligned* on a proper boundary (64, 128, 256 or 512-bit).

The compiler can generate automatically vector instructions when possible. A double precision AVX auto-vectorized loop generates 3 loops:

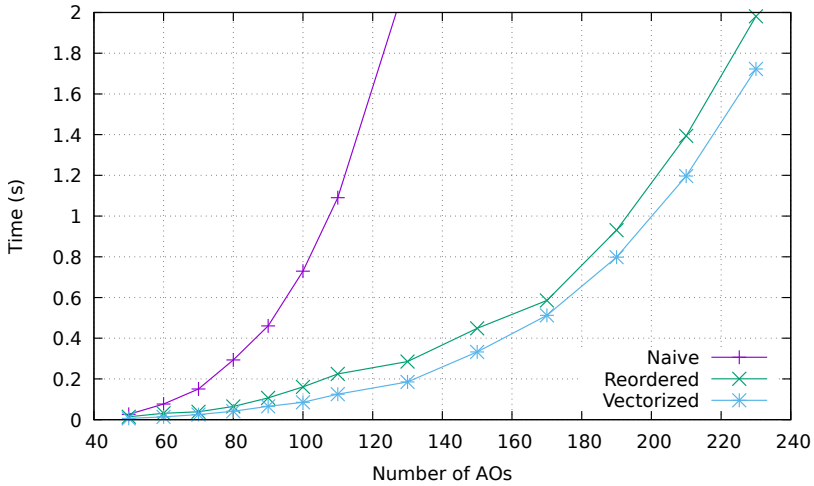
1. Peel loop (scalar) First elements until the 256-bit boundary is met
2. Vector loop Vectorized version until the last vector of 4 elements
3. Tail loop (scalar) Last elements

Most efficient for large loop counts.

Automatic vectorization



Example: Coulomb Matrix

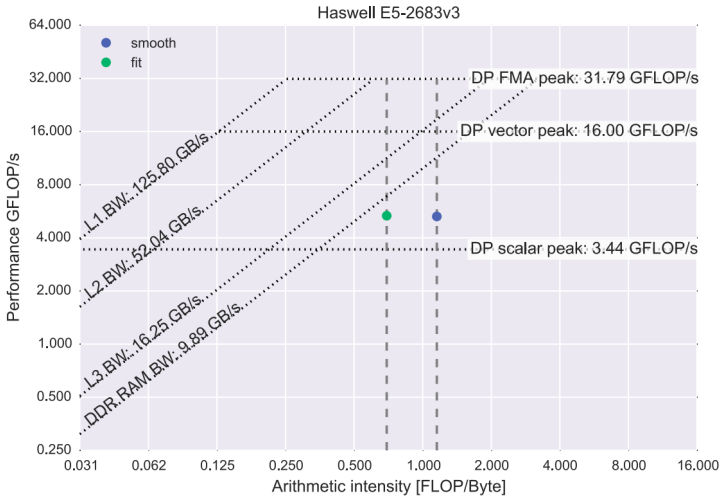


$$J_{\mu\nu} = \sum_{\lambda\sigma} P_{\lambda\sigma} \iint \chi_\mu(r_1)\chi_\lambda(r_2)\frac{1}{r_{12}}\chi_\nu(r_1)\chi_\sigma(r_2)dr_1 dr_2$$

```
1     do nu=1,N
2        do mu=1,N
3           J(mu,nu) = 0.d0
4           do si=1,N
5              do la=1,N
6                 J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
7              end do
8           end do
9        end do
10    end do
```

- 2 N^4 flops, N^2 stores (J) + N^2 (N^2 loads (P)) + N^4 loads (ERI)
- For N=150, P needs 176kB: fits in L2

Roofline model



Hugo, Perez, doi:10.1088/1742-6596/898/3/032052

```
1  do nu=1,N
2   do mu=1,N-4,4
3    J(mu:mu+3,nu) = 0.d0
4    do si=1,N
5     do la=1,N
6      J(mu+0,nu) = J(mu+0,nu) + P(la,si) * ERI(la,si,mu+0,nu)    ! P(la,si) is
7      J(mu+1,nu) = J(mu+1,nu) + P(la,si) * ERI(la,si,mu+1,nu)    ! re-used 4x &
8      J(mu+2,nu) = J(mu+2,nu) + P(la,si) * ERI(la,si,mu+2,nu)    ! 1 SIMD store
9      J(mu+3,nu) = J(mu+3,nu) + P(la,si) * ERI(la,si,mu+3,nu)    ! for J
10    end do
11   end do
12  end do
13  do mu=N-4+1,N
14   J(mu,nu) = 0.d0
15   do si=1,N
16    do la=1,N
17     J(mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
18    end do
19   end do
20  end do
21 end do
```

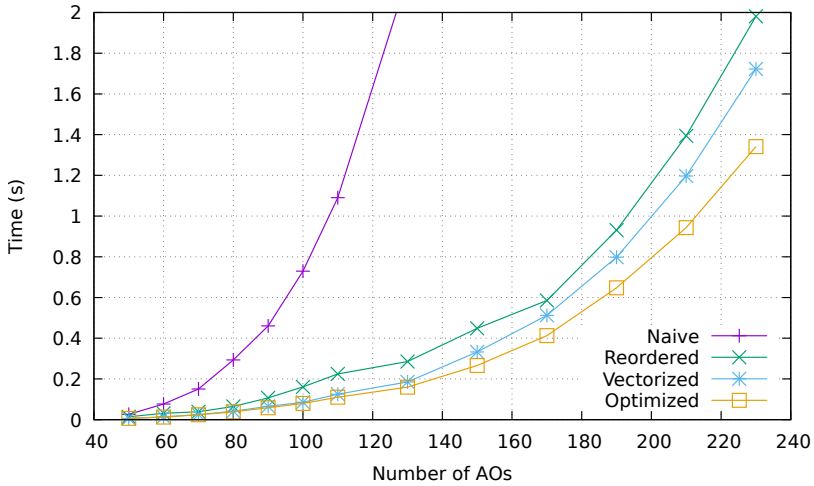
```
1       J     (mu,nu) = J(mu,nu) + P(la,si) * ERI(la,si,mu,nu)
```

- 2 loads
- 2 flops
- 2/16 = 0.125 flops/byte

```
1       J     (mu+0,nu) = J(mu+0,nu) + P(la,si) * ERI(la,si,mu+0,nu) ! P(la,si) is
2       J     (mu+1,nu) = J(mu+1,nu) + P(la,si) * ERI(la,si,mu+1,nu) ! re-used 4x &
3       J     (mu+2,nu) = J(mu+2,nu) + P(la,si) * ERI(la,si,mu+2,nu) ! 1 SIMD store
4       J     (mu+3,nu) = J(mu+3,nu) + P(la,si) * ERI(la,si,mu+3,nu) ! for J
```

- 1 load `P(la,si)` + 4 loads `ERI`
- 8 flops
- 8/40 = 0.2 flops/byte

Increasing arithmetic intensity



Matrix multiplications

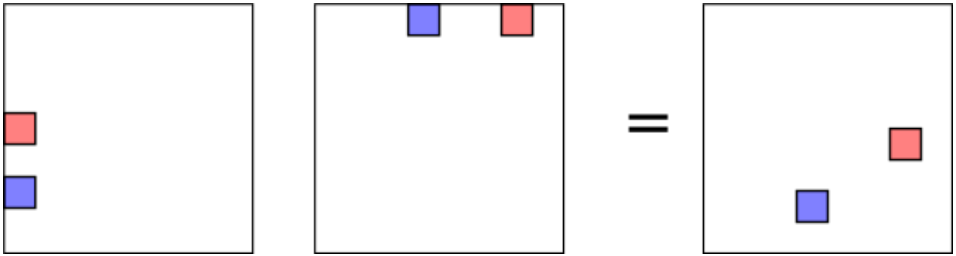
Why matrix multiplications?

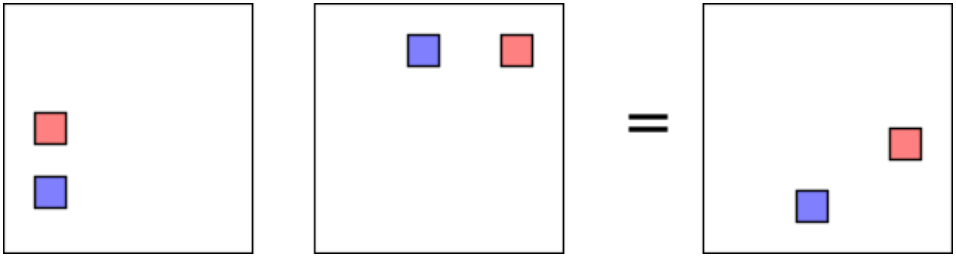
$$C_{mn} \leftarrow C_{mn} + A_{mk} \times B_{kn}$$

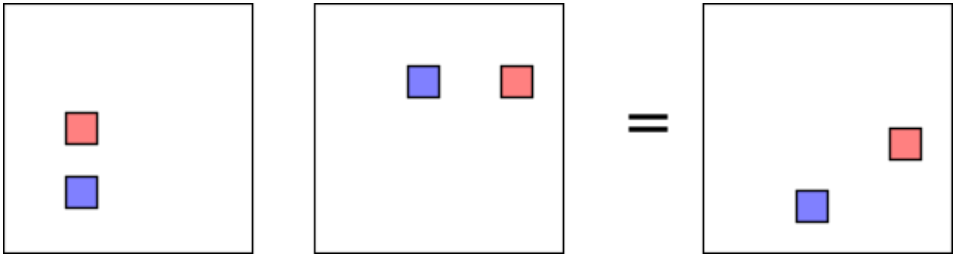
- Fused Multiply-Add (FMA) : $a \leftarrow a + b \times c$ is one SIMD instruction executed in 1 cycle
- A CPU has
  - Two independent SIMD FP units
  - Two independent SIMD memory load units
  - One memory store unit
- MM can be easily parallelized by performing the product in blocks

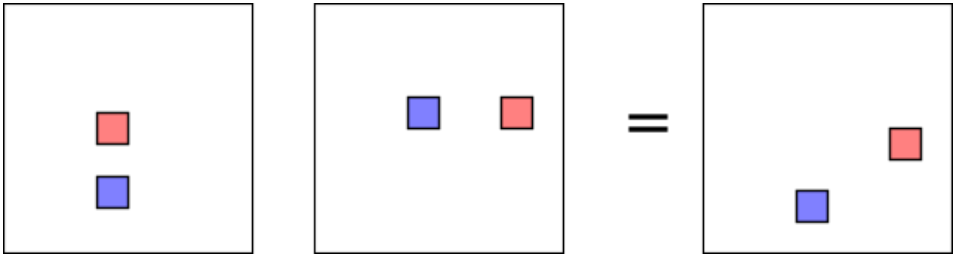
Arithmetic intensity

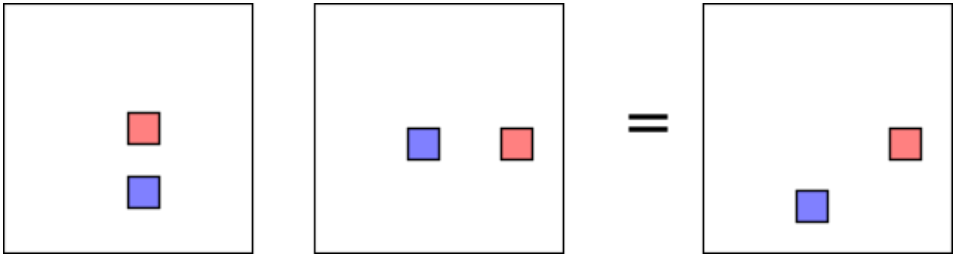
- Both CPUs and GPUs are well adapted compute matrix multiplications.
- DGEMM with vendor libraries can reach:
  - 90% of the peak on CPU, 80% of the peak on GPU

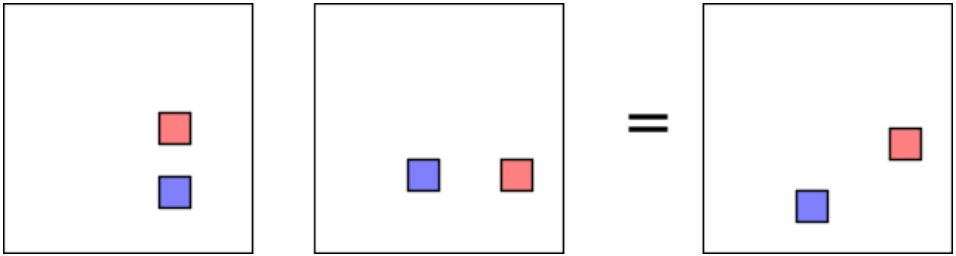


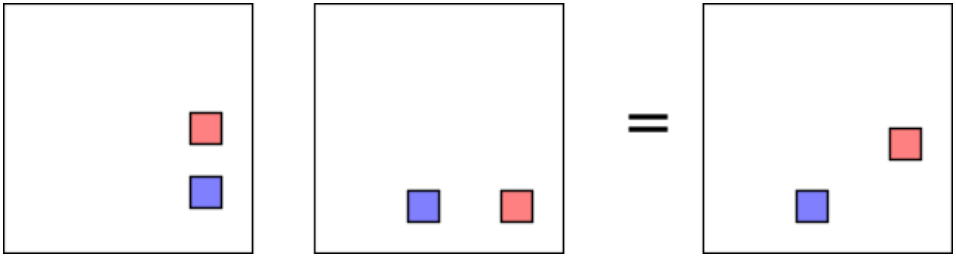


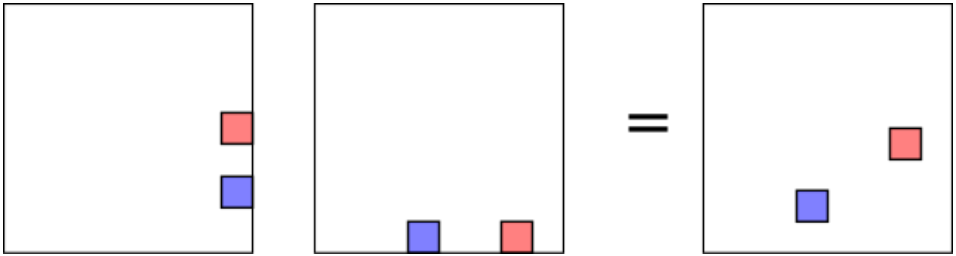












A vector / matrix and an array should not be confused:

- Vector / Matrix: Mathematical objects, with *logical* dimensions
- Array: a collection of same type data items that can be selected by indices. Dimensions are *physical*: they correspond to addresses in memory.

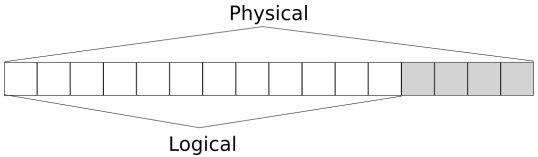
A matrix is stored in an array of FP numbers. The array has to be big enough to accept the matrix in it.

```
1       double precision :: A(nmax)
2       do i=1,n
3          A(i) = ...
4       end do
```

- Vector : a_i, $1 \leq i \leq n$
- Array : Allocated memory: `A(1:nmax)`

```
1   double* A;
2   A = malloc (sizeof(double) * nmax);
```

a_i = `A(i) = A[i-1]`



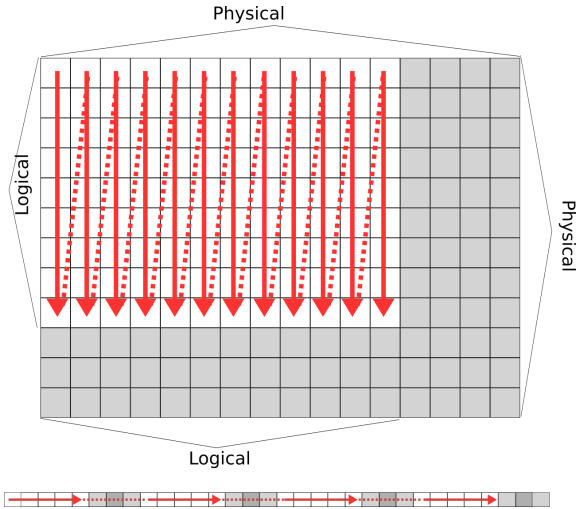
```
1       double precision :: A(mmax,nmax)
2       do j=1,n
3          do i=1,m
4             A(i,j) = ...
5          end do
6       end do
```

- Matrix : A_{ij}, $1 \leq i \leq m$, $1 \leq j \leq n$
- Array : Allocated memory: `A(1:mmax,1:nmax)`, `&(A[0])`

```
1   double* A;
2   A = malloc (sizeof(double) * nmax * mmax);
```

A_{ij} = `A(i,j) = A[ (j-1)*mmax + i-1 ]`

Example: Matrix



Decrypting BLAS DGEMM arguments

$$C'_{mn} = \beta C_{mn} + \alpha \sum_k A_{mk} B_{kn}$$

- DGEMM : Double precision GEneral Matrix Matrix multiplication
- `LDA` : Leading dimension of `A`

Why pass `LDA` ? To compute the address in the array.

```
1       double precision :: A(LDA,*)
2       double precision :: B(LDB,*)
3       double precision :: C(LDC,*)
4
5       call DGEMM( 'N', 'N'   & ! Transposed?
6            , m, n, k         & ! Dimensions
7            , 1.d0            & ! alpha
8            , A, size(A,1)    & ! A
9            , B, size(B,1)    & ! B
10           , 0.d0            & ! beta
11           , C, size(C,1) )    ! C
```

Finding matrix multiplications by reshaping

- A rank-2 array can be reshaped into a rank-1 array
- A matrix can be interpreted as a vector
- A rank-3 array can be reshaped into a rank-2 array

```
1        do j=1,n
2           do i=1,n
3              C(i,j) = 0.d0
4              do l=1,n
5                 do k=1,n
6                    C(i,j) = C(i,j) + A(k,l,i) * B(k,l,j)
7                 end do
8              end do
9           end do
10       end do
```

- `A(k,l,i)` can be reshaped as: `A(kl,i)`
- `B(k,l,j)` can be reshaped as: `B(kl,j)`

This can be computed with a matrix multiplication:

$$C_{ij} = \sum_{kl} A^{\dagger}_{kl,i} B_{kl,j}$$

```
1       call DGEMM('T', 'N', n, n, (n*n), 1.d0, &
2                  A, size(A,1)*size(A,2), &
3                  B, size(B,1)*size(B,2), 0.d0, &
4                  C, size(C,1) )
```

$$
\begin{aligned}
& \int \phi_\alpha(r_1)\phi_\beta(r_2)\frac{1}{|r_1 - r_2|}\phi_\gamma(r_1)\phi_\delta(r_2)\mathrm{d}r_1\mathrm{d}r_2 \\
= & \sum_{ijkl} B_{i\alpha}B_{j\beta}B_{k\gamma}B_{l\delta}\int \chi_i(r_1)\chi_j(r_2)\frac{1}{|r_1 - r_2|}\chi_k(r_1)\chi_l(r_2)\mathrm{d}r_1\mathrm{d}r_2 \\
C_{\alpha\beta\gamma\delta} = & \sum_{ijkl} A_{ijkl}\cdot B_{i\alpha}\cdot B_{j\beta}\cdot B_{k\gamma}\cdot B_{l\delta}
\end{aligned}
$$

- A : N^4 integrals in AO basis
- C : M^4 integrals in MO basis
- Conventional algorithm : Transform indices one by one : M^5 scaling

Example: 4-index transformation (Loops)

```
1       C=0.d0
2       do s=1,N
3          do l=1,N
4             do k=1,N
5                do j=1,N
6                   do i=1,N
7                      C(i,j,k,s) = C(i,j,k,s) + A(i,j,k,l) * B(l,s)
8                   end do
9                end do
10            end do
11         end do
12      end do
```

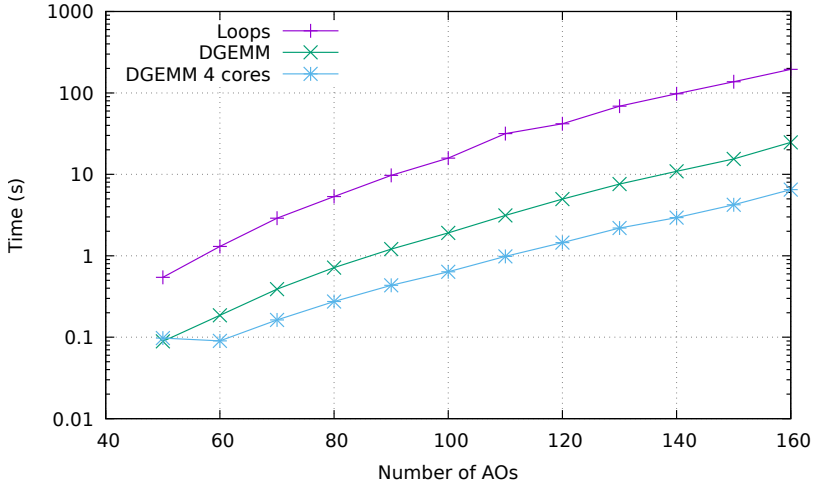
- Do one of such loop for each index

Example: 4-index transformation (DGEMM)

$$
\begin{aligned}
P_{jkl,\alpha} &\leftarrow \textstyle\sum_l A_{i,jkl} B_{i,\alpha} && \mathrm{P} = \mathrm{A}^{\dagger}.\mathrm{B} \\
Q_{kl\alpha,\beta} &\leftarrow \textstyle\sum_l P_{j,kl\alpha} B_{j,\beta} && \mathrm{Q} = \mathrm{P}^{\dagger}.\mathrm{B} \\
R_{l\alpha\beta,\gamma} &\leftarrow \textstyle\sum_l P_{k,l\alpha\beta} B_{k,\gamma} && \mathrm{R} = \mathrm{Q}^{\dagger}.\mathrm{B} \\
C_{\alpha\beta\gamma,\delta} &\leftarrow \textstyle\sum_l R_{l,\alpha\beta\gamma} B_{l,\delta} && \mathrm{C} = \mathrm{R}^{\dagger}.\mathrm{B}
\end{aligned}
$$

```
1  call DGEMM('T','N', (nao*nao*nao), nmo, nao, 1.d0, A, nao, B, nao, 0.d0, P, (nao*nao*nao))
2  call DGEMM('T','N', (nao*nao*nmo), nmo, nao, 1.d0, P, nao, B, nao, 0.d0, Q, (nao*nao*nmo))
3  call DGEMM('T','N', (nao*nmo*nmo), nmo, nao, 1.d0, Q, nao, B, nao, 0.d0, R, (nao*nmo*nmo))
4  call DGEMM('T','N', (nmo*nmo*nmo), nmo, nao, 1.d0, R, nao, B, nao, 0.d0, C, (nmo*nmo*nmo))
```

Example: 4-index transformation



APPENDIX: DENSITY MATRICES AND ORBITAL HESSIAN

For completeness, we include here expressions for the pCCD density matrices and orbital rotation Hessian; together with the orbital rotation gradient of Eq. (25), these provide everything needed for the Newton-Raphson algorithm we use for orbital optimization.

Recall that the energy is written as

$$\mathcal{E}(\kappa) = \langle 0|(1+Z)\,\mathrm{e}^{-T}\,\mathrm{e}^{-\kappa}\,H\,\mathrm{e}^{\kappa}\,\mathrm{e}^{T}|0\rangle \tag{A1}$$

with

$$\kappa = \sum_{p>q}\sum_{\sigma}\kappa_{pq}\left(c^{\dagger}_{p_\sigma}c_{q_\sigma} - c^{\dagger}_{q_\sigma}c_{p_\sigma}\right), \tag{A2}$$

where the orbital rotation is given by the unitary transformation $\exp(\kappa)$. At every step of the Newton-Raphson scheme,

$$\left.\frac{\partial\mathcal{E}(\kappa)}{\partial\kappa_{pq}}\right|_{\kappa=0} = \mathcal{P}_{pq}\sum_{\sigma}\langle[H, c^{\dagger}_{p_\sigma}c_{q_\sigma}]\rangle, \tag{A3}$$

where $\mathcal{P}_{pq}$ is a permutation operator $\mathcal{P}_{pq} = 1-(p\leftrightarrow q)$ and the notation for the expectation value means

$$\langle O\rangle = \langle 0|(1+Z)\,\mathrm{e}^{-T}\,O\,\mathrm{e}^{T}|0\rangle. \tag{A4}$$

Similarly, the Hessian is

$$\begin{aligned} H_{pq,rs} &= \left.\frac{\partial^2\mathcal{E}(\kappa)}{\partial\kappa_{pq}\,\partial\kappa_{rs}}\right|_{\kappa=0} \\ &= \frac{1}{2}\mathcal{P}_{pq}\mathcal{P}_{rs}\sum_{\sigma,\eta}\langle[[H, c^{\dagger}_{p_\sigma}c_{q_\sigma}], c^{\dagger}_{r_\eta}c_{s_\eta}]\rangle \\ &\quad + \frac{1}{2}\mathcal{P}_{pq}\mathcal{P}_{rs}\sum_{\sigma,\eta}\langle[[H, c^{\dagger}_{r_\eta}c_{s_\eta}], c^{\dagger}_{p_\sigma}c_{q_\sigma}]\rangle, \end{aligned} \tag{A5}$$

where η is another spin index. We obtain

$$\begin{aligned} H_{pq,rs} = \mathcal{P}_{pq}\mathcal{P}_{rs}\Bigg\{ &\frac{1}{2}\sum_{u}\Big[\delta_{qr}\left(h^{u}_{p}\gamma^{s}_{u} + h^{s}_{u}\gamma^{u}_{p}\right) + \delta_{ps}\left(h^{u}_{r}\gamma^{q}_{u} + h^{q}_{u}\gamma^{u}_{r}\right)\Big] - \left(h^{s}_{p}\gamma^{q}_{r} + h^{q}_{r}\gamma^{s}_{p}\right) \\ &+ \frac{1}{2}\sum_{tuv}\Big[\delta_{qr}\left(v^{uv}_{pt}\Gamma^{st}_{uv} + v^{st}_{uv}\Gamma^{uv}_{pt}\right) + \delta_{ps}\left(v^{qt}_{uv}\Gamma^{uv}_{rt} + v^{uv}_{rt}\Gamma^{qt}_{uv}\right)\Big] \\ &+ \sum_{uv}\left(v^{uv}_{pr}\Gamma^{qs}_{uv} + v^{qs}_{uv}\Gamma^{uv}_{pr}\right) - \sum_{tu}\left(v^{st}_{pu}\Gamma^{qu}_{rt} + v^{ts}_{pu}\Gamma^{qu}_{tr} + v^{qu}_{rt}\Gamma^{st}_{pu} + v^{qu}_{tr}\Gamma^{ts}_{pu}\right)\Bigg\}. \end{aligned} \tag{A6}$$

Henderson *et al*, http://dx.doi.org/10.1063/1.4904384

Take-home messages

- Organize data for stride-1 access
- Favor good access to writing
- Consider matrix multiplication as *the magic instruction for performance*
- Porting code based on DGEMM to GPU can be done with minimal effort: use MAGMA instead of MKL