Precision Limits and Scaling Performance in Fortran Numerical Computations

- Alberto Salvador
- Physics of Data, University of Padua

- Course: Quantum information and computing
- Assignment 1

28 October 2024

Number precision (in Fortran)

Fortran data types:

- Integers
- Real
- Complex
- Character
- …

The numbers stored have **finite precision**, which can be associated during their declaration.

INTEGER*2 (4): allocates 2 (4) bytes of memory for storing a signed integer

REAL(4) (8) have a precision up to 8 (16) digits: all the digits that comes after are meaningless for the computation and put to zero

To investigate finite precision:

(a) Sum 2.000.000 and 1 using INTEGER*2 and INTEGER*4

(b) Sum pi*10^32 and (2^0.5)*10^21 using single and double precision

result

```
root@LAPTOP-9GNANRQ5:/home/albertos/quantumInfo/ex1# gfortran -fno-range-check number_precision.f90
root@LAPTOP-9GNANRQ5:/home/albertos/quantumInfo/ex1# ./a.out
 The sum of 2000000 and 1 with 2-byte precision is:  -31615
 The sum of 2000000 and 1  with 4-byte precision is:      2000001

 The sum of pi*10^32 and sqrt(2)*10^21 with single precision is:    3.14159278E+32
 The sum of pi*10^32 and sqrt(2)*10^21 with single precision is:    3.1415926536039354E+032
```

Matrix multiplication (in Fortran)

Matrix matrix multiplication is defined in linear algebra by the well-known “**row-by-columns**” algorithm

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} \xrightarrow{\text{C = AB}} \mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

with algorithmic complexity: O(n^3)

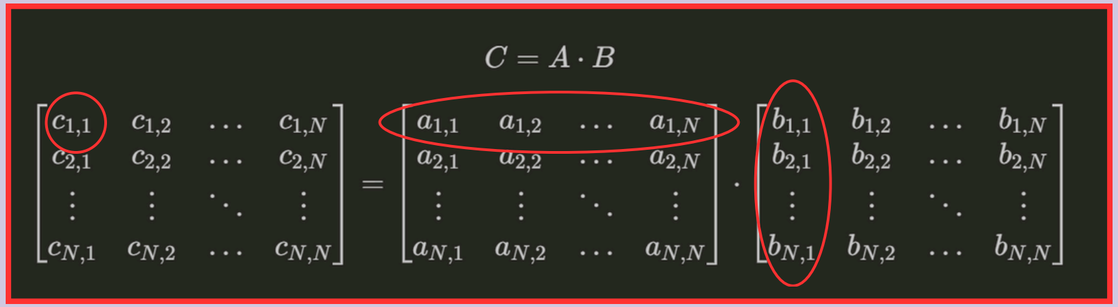
There exist different ways of reducing the algorithmic complexity of this operation (e.g. with the *Cooper-Winegard* algorithm we get up to O(n^2.376)) . These methods are already implemented by default in programming languages as Fortran.

Additional improvements can be achieved by considering how matrices are allocated in memory by Fortran.
More specifically, columns are stored contiguously, so transforming the matrix A so that the algorith becomes a “column-by-column” one could be beneficial.

Matrix multiplication methods

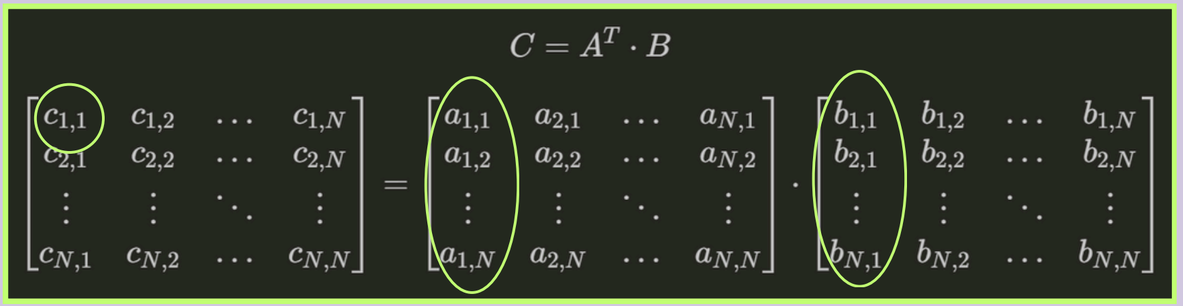
Method 1 (row-by-col)

```
do i=1, N
    do j = 1, N
        do k = 1, N
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
        end do
    end do
end do
```



Method 2 (col-by-col)

```
do i=1, N
    do j = 1, N
        do k = 1, N
            C(i,j) = C(i,j) + A_T(k,i)*B(k,j)
        end do
    end do
end do
```



Method 3 (built-in function)

```
C = matmul(A,B)
```

Optimized compiling

Flags utilized:

```
root@LAPTOP-9GNANRQ5:/home/albertos# gfortran -O3 -march=native
 -ftree-vectorize -funroll-loops -floop-block matrix_utils.f90
matmatmul.f90
```

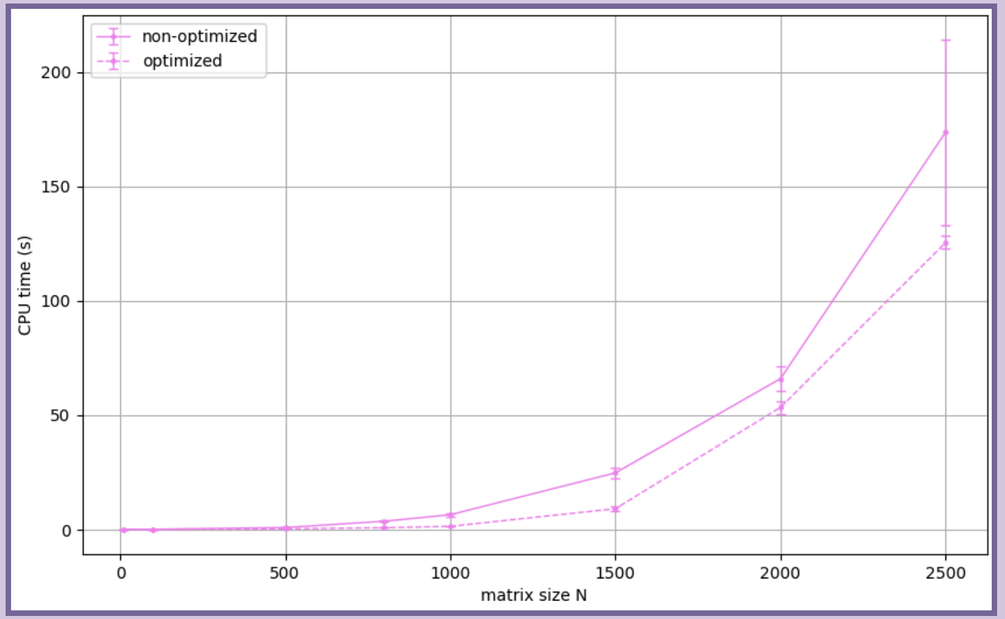
-O3:
general optimization: reduction of code size and execution time

-march=native:
code optimization for the specific machine's architecture

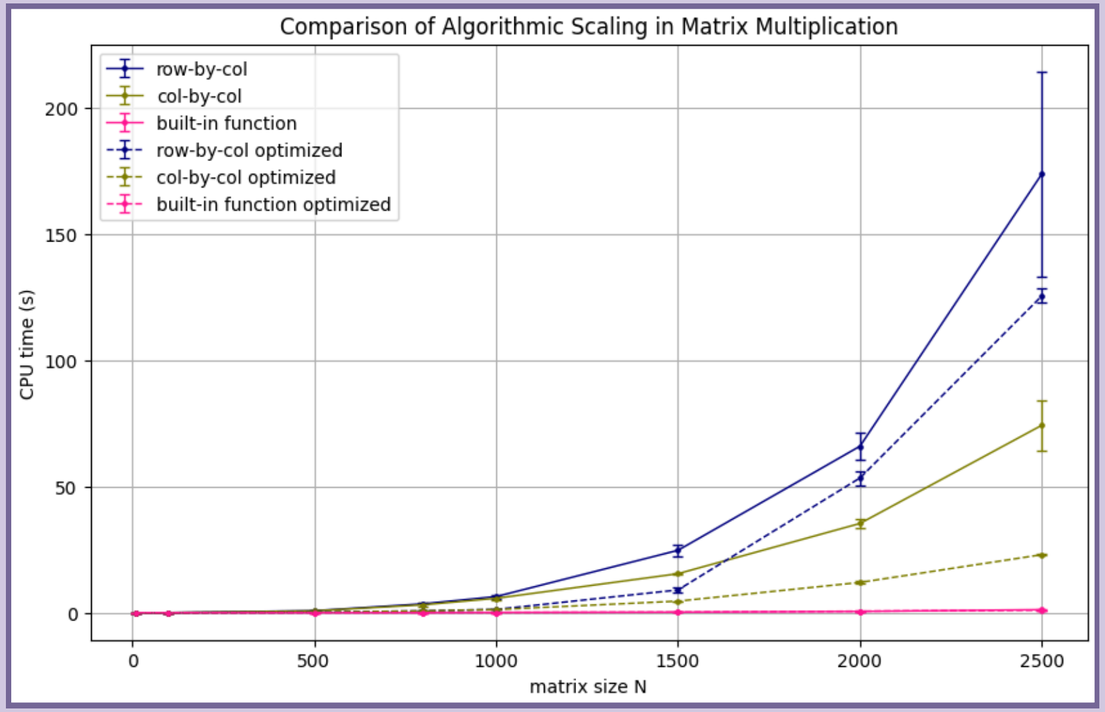
-ftree-vectorize:
automatic vectorization from the compiler

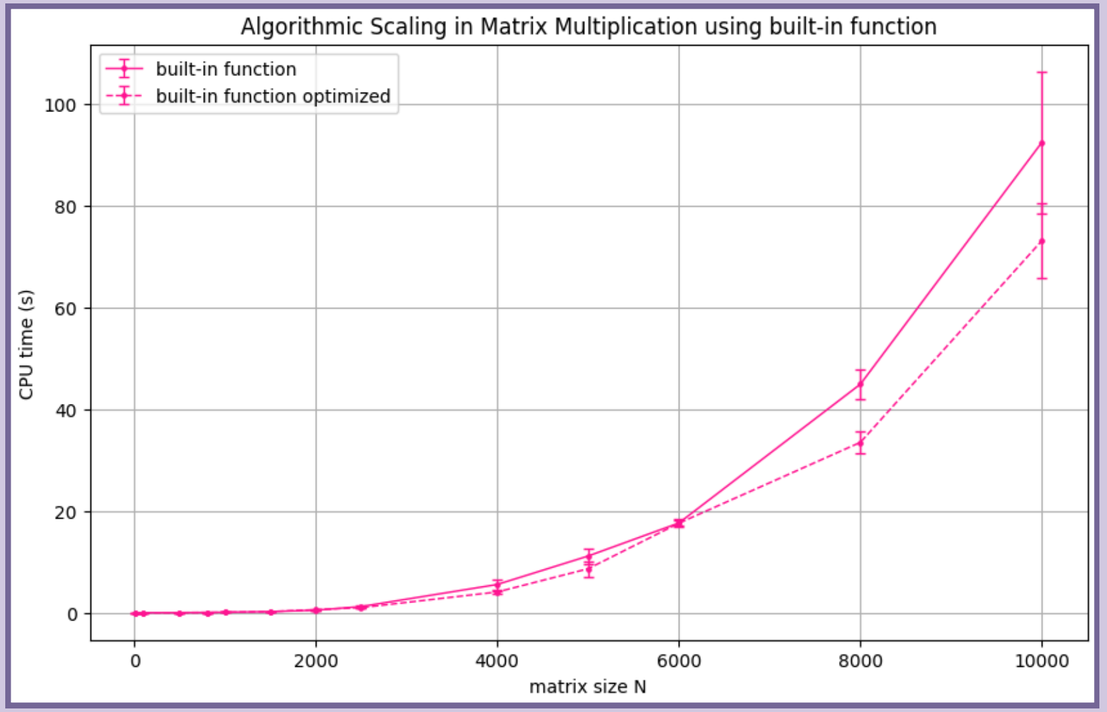
-funroll-loops:
optimization of do loops by unrolling them

-floop-block:
optimization of for loops



Testing the performances





Built-in method is the most advantageous and should be the preferred anytime.

Clearly gain in performance using the optimization flags for all cases.