Quantum Information and Computing 2020/21
Week 4 report

Rocco Ardino
(Dated: Monday 2nd November, 2020)

Studying the performances of an algorithm, depending on some input variables, requires a certain level of automatization. This task can be easily accomplished using scripting languages such as Python. In this work, we take as algorithm the square matrices multiplication, whose CPU time consumption depends on the matrices size. The results are then stored, fitted and compared through meaningful plots with a model based on computational complexity considerations.

1 Theory

Let us consider two matrices $A \in \mathbb{R}^{n\times m}$ and $B \in \mathbb{R}^{p\times q}$, of dimensions respectively $n \times m$ and $p \times q$. If $m = p$, it is possible to compute the matrix product AB and the result is a matrix $C \in \mathbb{R}^{n\times q}$ of dimensions $n \times q$, with matrix elements:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \qquad \begin{matrix} i=1,\dots,n \\ j=1,\dots,q \end{matrix} \quad . \tag{1}$$

This algorithm is implemented in a very simple way through:

- a loop over k for the calculation of a single matrix element (complexity: $\mathcal{O}(m = p)$);
- two loops over i and j for the calculation of all the entries of the product matrix (complexity: $\mathcal{O}(nq)$)

Therefore, the overall complexity of the operation is $\mathcal{O}(nmq)$, being $\mathcal{O}(n^3)$ for the special case of two square matrices multiplication.

From a mathematical point of view, the order of the three loops is irrelevant for the result. However, it affects the performances depending on the way a language stores variables in memory.

2 Code Development

The modules **matrixmod** and **err_handling_mod** from the previous weeks are employed in this work along with a Python script for automatization of the study. The code for matrix matrix multiplication implemented in Fortran is explained in Subsection 2.1. The core functionalities of the Python script are explained in Subsection 2.2.

2.1 Matrix matrix multiplication algorithms

Matrix matrix multiplication algorithms have several possible implementations, depending on the order of the loops needed to access to the memory. For our study, we limit to two possible orders, corresponding to two user-defined functions:

- **matmul_col(mat1,mat2)**: the loop over consecutive elements of memory is the inner one. Its implementation is skectched in Listing 1;
- **matmul_row(mat1,mat2)**: the loop over consecutive elements of memory is the outer one. Its implementation is skectched in Listing 2.

Concerning their performances, the first method is expected to run faster since it exploits the rapid access to consecutive memory elements. However, the intrinsic Fortran implementation of `MATMUL` outperforms both the implementations, so all these three possibilities will be compared in the final results.

```
function matmul_col(mat1, mat2) result(res)
    ...

    ! mat1 and mat2 dimensions
    N = size(mat1, 1)
    M = size(mat1, 2)
    P = size(mat2, 1)
    Q = size(mat2, 2)

    ! pre-conditions and debug checks:
    ! * if input matrices have valid dimensions
    ! * if input matrices are multiplicable
    ...

    ! loop for matrix matrix multiplication
    do jj=1,Q
        do kk=1,M
            do ii=1,N
                res(ii,jj) = res(ii,jj) + mat1(ii,kk)*mat2(kk,jj)
            end do
        end do
    end do
end function
```

Listing 1. Implementation of the first algorithm of matrix matrix multiplication `matmul_col`.

```
function matmul_row(mat1, mat2) result(res)
    ...

    ! mat1 and mat2 dimensions
    N = size(mat1, 1)
    M = size(mat1, 2)
    P = size(mat2, 1)
    Q = size(mat2, 2)

    ! pre-conditions and debug checks:
    ! * if input matrices have valid dimensions
    ! * if input matrices are multiplicable
    ...

    ! loop for matrix matrix multiplication
    do ii=1,N
        do kk=1,M
            do jj=1,Q
                res(ii,jj) = res(ii,jj) + mat1(ii,kk)*mat2(kk,jj)
            end do
        end do
    end do
end function
```

Listing 2. Implementation of the second algorithm of matrix matrix multiplication `matmul_row`.

The three functions `matmul_col`, `matmul_row` and `MATMUL` are called in a test program, which takes as input command-line argument an integer representing the dimension of two square matrices to multiply. The way the input argument is taken and handled is sketched in Listing 3. Then, the CPU time needed for the operation is computed for every implementation and printed on the standard output.

```
! square matrices dimension
integer(4) :: N

! get command-line arguments (square matrix size)
character(len=:), allocatable :: argument
integer*4 :: arglen
call GET_COMMAND_ARGUMENT(1, length=arglen)
allocate(character(arglen) :: argument)
call GET_COMMAND_ARGUMENT(1, value=argument)

! convert command line argument to integer (matrix size)
read(argument(:),'(i5)') N
```

Listing 3. Handling of square matrix size n input argument.

2.2 Python script for automatization

To test the performances of the three algorithms depending on the matrices size, it is convenient to run the Fortran executable from a Python script for different values of n. For every run, the script takes the standard output with CPU time results and converts it to three numerical values, one for every algorithm. This trick is described in Listing 4.

```
import numpy as np
import os
import subprocess

# function for converting fortran output into numerical results
def get_output(cmd, N):
    results = subprocess.run([cmd, str(N)], stdout=subprocess.PIPE)
    results = results.stdout.decode('utf-8').strip().split('\n')
    results = [float(res.strip()) for res in results]
    return np.array(results)
```

Listing 4. Python function to convert standard output of Fortran executable to CPU time numerical values.

Therefore, the script in order:

- launches the Fortran executable m times for every single value of n;
- gets the CPU time for each of the m samples;
- takes the mean value and the standard deviation over the m samples;
- does the previous steps for every algorithm and stores the respective results in .dat files.

These steps are described in Listing 5.

```
# No optimization flags
for j in range(len(Oflags)):
    Oflag = Oflags[j]
    print("["+Oflag+"] : start ------------------------------------------")
    os.system(comp + Oflag + opt + exec + src)
    for N,i in zip(Ns,range(len(Ns))):
        print("Running matmul functions for N = " + str(N))
        res = np.zeros((3,M))
        for k in range(M):
            res[:,k] = get_output(cmd, N)
        # store mean
```

```
        matmul_col_data[i,j] = np.mean(res[0,:])
        matmul_row_data[i,j] = np.mean(res[1,:])
        matmul_data[i,j]     = np.mean(res[2,:])
        # store standard deviation
        matmul_col_data[i,j+len(Oflags)] = np.std(res[0,:]) / np.sqrt(M)
        matmul_row_data[i,j+len(Oflags)] = np.std(res[1,:]) / np.sqrt(M)
        matmul_data[i,j+len(Oflags)]     = np.std(res[2,:]) / np.sqrt(M)
    print("["+Oflag+"] : end ----------------------------------------------")
```

Listing 5. Sketch of code for scan on the square matrices size n.

Lastly, the output files are used to plot the results through a gnuplot script. Moreover, the ladder performs several fits for every dataset.

3 Results

The CPU time of the three algorithms under study is monitored for n spanning between $N_{\text{min}} = 50$ and $N_{\text{max}} = 2000$. In particular, 30 values of n are chosen in this interval using a log-spacing, since the plots of the results are showed in log-log scale. The number of samples m for the mean value and standard deviation computation is fixed to 5. So, the results without the use of optimization flags at compile time are displayed in Figure 1.

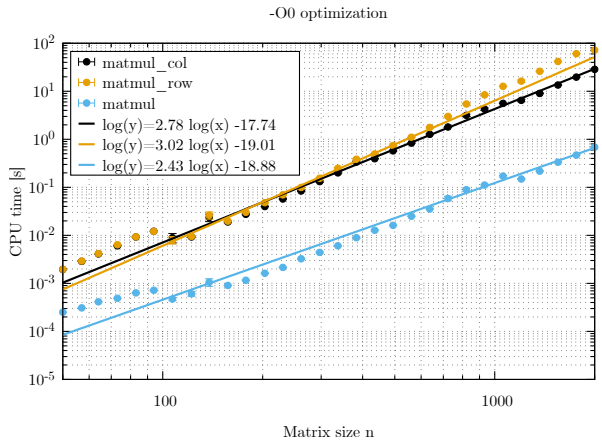
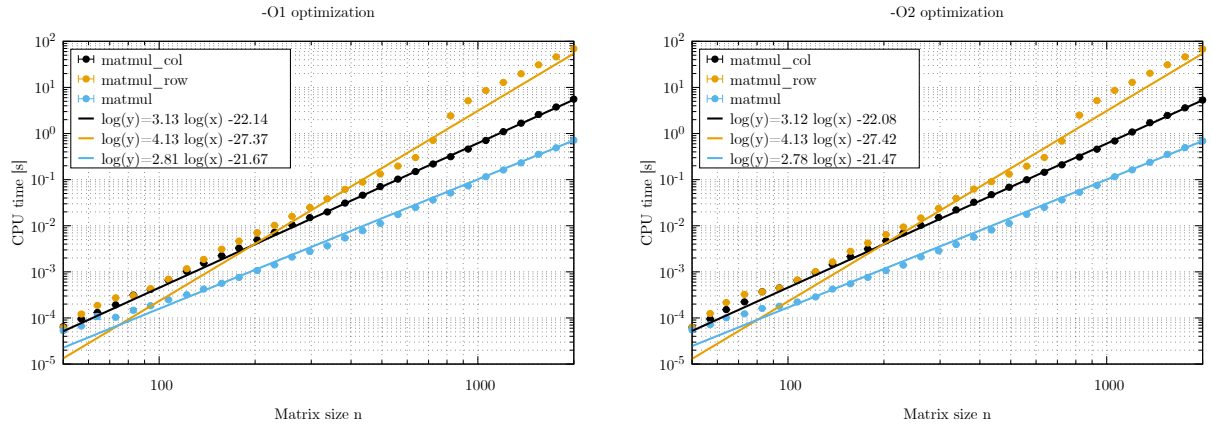


Figure 1. Performances of the two implementations and of the intrinsic function depending on the matrix size n, without using optimization flags.

Moreover, the performances of the algorithms are analyzed for different optimization flags added at compile time, such as `-O1`, `-O2`, `-O3` and `-Ofast`. For these cases, the results are showed respectively in Figures 2a, 2b, 2c and 2d. Lastly, every result for the slope coefficient obtained by fit is reported in Table 1 along with the associated uncertainty.

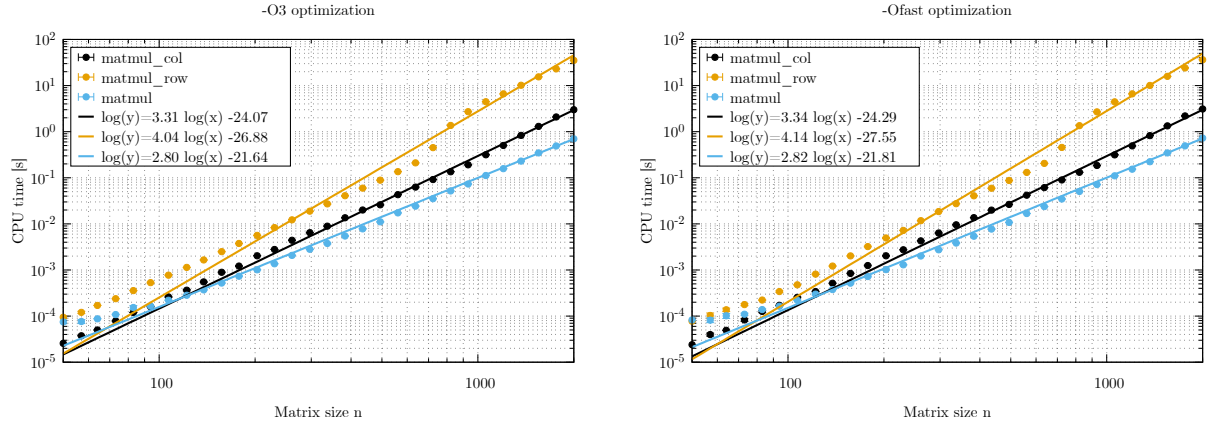
As we can see from the plot without optimization flags (`-O0`), the results for `matmul_row` is in agreement with an expected slope coefficient of ~ 3, showing however some fluctuations. The slope for `matmul_col` and, in particular, for the intrinsic `matmul` are significantly smaller, due to a more efficient implementation.

Concerning the cases with optimization flags, we can see from the plots how the results change, showing a deviation from the expected slope for `matmul_row`. Insted, `matmul_col` still agrees with the expectation, while `matmul` shows worse results with respect to the case of no optimization flags but still below the complexity of $\mathcal{O}(n^3)$.



(a) **-O1**

(b) **-O2**



(c) **-O3**

(d) **-Ofast**

Figure 2. Comparison of performances depending on n for different optimization flags.

Flag	$\mathbf{m}_{\texttt{matmul\_col}}$ $[\mathrm{s}^{-1}]$	$\mathbf{m}_{\texttt{matmul\_row}}$ $[\mathrm{s}^{-1}]$	$\mathbf{m}_{\texttt{MATMUL}}$ $[\mathrm{s}^{-1}]$
-O0	2.78 ± 0.05	3.02 ± 0.08	2.43 ± 0.05
-O1	3.14 ± 0.02	4.1 ± 0.2	2.81 ± 0.03
-O2	3.12 ± 0.03	4.1 ± 0.2	2.78 ± 0.03
-O3	3.31 ± 0.04	4.0 ± 0.2	2.80 ± 0.03
-Ofast	3.34 ± 0.05	4.1 ± 0.2	2.83 ± 0.03

Table 1. Slope coefficients for every case, obtained from fit results.

4 Self-evaluation

The implementation of the code has led to interesting results. Deviations from the expected behaviour are found for both implementations in different cases. These observations require further studies, possibly scanning larger values of n, in order to find a valid explanation. However, this cannot actually be done because of the insufficient computational power of the machine on which the code is executed.