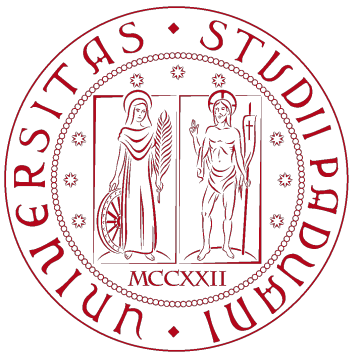
Università degli studi di Padova

Quantum Information and Computing
Prof. Simone Montangero

Exercise 01

Author:
Alessandro Marcomini (2024286)

a.a 2021/2022



Università
degli Studi
di Padova

Exercise 1: setup

1. Create a working directory
2. Open emacs and write your first program in FORTRAN
3. Submit a test job
4. (Optional) Connect to the cluster spiro.fisica.unpd.it via ssh and repeat the execution

Solution

(1) I set up a repo on my GitHub account to contain all the code I wrote for this set of exercises and the upcoming ones.
(2,3) As a first program in FORTRAN I created the most known and basic program of all time - "HelloWorld". This was indeed the occasion to familiarize with the syntax and the "gfortran" compiler.
(4) As for this point, since I do not have access to a remote machine on the cloud, I just accessed the gate as reported in the figure below. From here, I could simply access the IP of the virtual machine provided to me and perform computation on there.

```
ale@MacBook-Pro-di-Alessandro ~ % ssh gate
amarcomi@gate.cloudveneto.it's password:
Last login: Tue Nov  2 23:04:03 2021 from host-82-60-121-100.retail.telecomitalia.it
bash-4.2$
```

Exercise 2: Number precision

Integer and real numbers have a finite precision. Explore the limits of INTEGER and REAL in Fortran.

1. Sum the numbers $2'000'000$ and 1 with INTEGER*2 and INTEGER*4
2. Sum the numbers $\pi \cdot 10^{32}$ and $\sqrt{2} \cdot 10^{21}$ in single and double precision

Solution

I wrote some simple FORTRAN code (IntSum, point (1) and DoubleSum, point (2)) to investigate the limits of this numerical types. Results are printed and shown below.

```
ale@MBPdiAlessandro Ex01 % ./IntSum.out
 2.000.000 + 1 (integer*2) =  -31615
 2.000.000 + 1 (integer*4) =      2000001
ale@MBPdiAlessandro Ex01 % ./DoubleSum.out
 order(E32) + order(E21) (real*4) =    3.14159278E+32
 order(E32) + order(E21) (real*8) =    3.1415926536039354E+032
 Result difference smaller than 1E-12: T
ale@MBPdiAlessandro Ex01 %
```

As one can see, the integer sum presents a clear case of overflow when the input is larger than the memory allocated. In fact, with INTEGER*2 type the maximum writable integer is $\mathcal{O}(10^5) < 2'000'000$ (I also had to pass the "-fno-range-check" flag to make the code compile). As a result, we have a misclassification of the sign bit that results in a negative outcome. On the other hand, we can see that the operation on real numbers (where I defined $\pi \equiv 4 \cdot \arctan(1)$) stays comfortably inside the single precision (large compatibility between the results).

Exercise 3: Test performance

Matrix matrix multiplication is many times the bottleneck of linear algebra computations.

1. Write explicitely the matrix-matrix multiplication loop in two different orders

2. Use the FORTRAN intrinsic function
3. Increase the matrix size and use the FORTRAN Function CPUTIME to monitor the code performance
4. Use the compiler different optimization flags and monitor the performances

Solution

(1) I considered the case for $A, B \in \mathcal{M}_{n\times n}$. Recalling that:

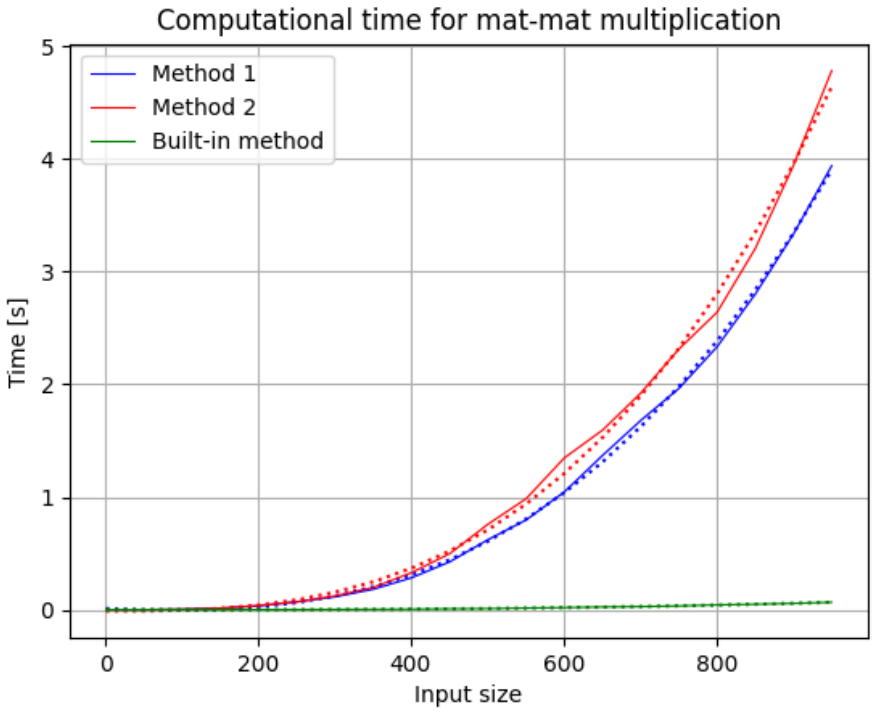
$$C = AB \iff C_{ij} = \sum_k A_{ik}B_{kj}$$

I exploited nested FOR loops with running indexes i, j, k in the range $1, \ldots, n$, iterating in one case on A rows (B columns) and in the other case vice versa:

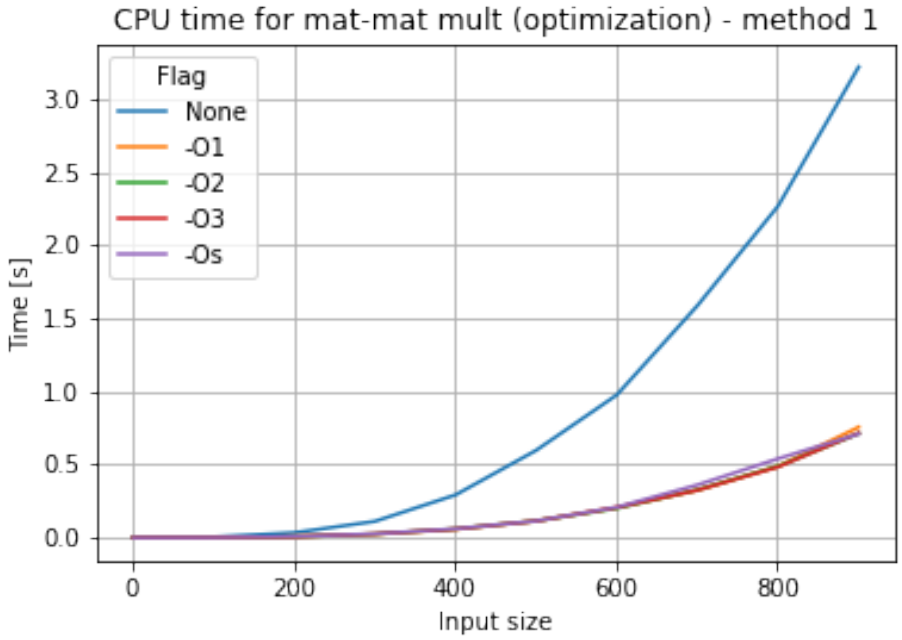
```
! FIRST METHOD

do I = 1,size
    do J = 1,size
        do K = 1, size
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
        end do
    end do
end do
```

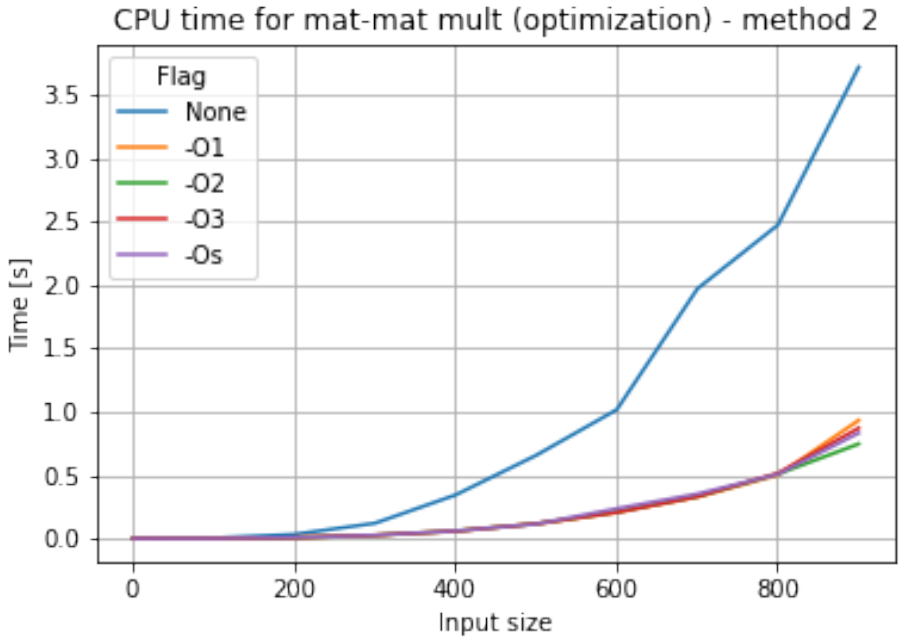
```
! SECOND METHOD

do J = 1,size
    do I = 1,size
        do K = 1, size
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
        end do
    end do
end do
```

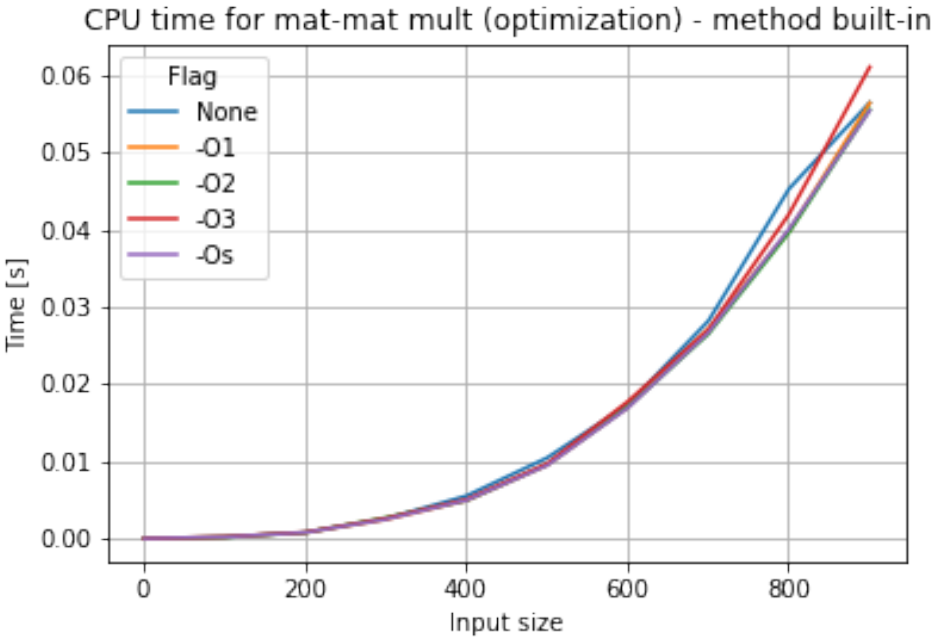
(2) Instead, I could achieve the same result by the means of the FORTRAN function "MATMUL(A,B)".
(3) I tested the performance of these methods creating subroutines and organizing the program so to reduce at minimum the lines of code and the memory usage. The results are reported in the plot below



As we can see from the plot above, where I increased the input size by 50 each time, the built-in method scales significantly better than the handwritten ones. Moreover, it is worth noticing as swapping the two loops is not indeed a truly symmetrical operation, since the CPU usage time grows differently. I personally believe that this has to do with the cashing procedure of the machine and it makes sense to have different computational time since this becomes a non-trivial factor for large matrices ($\mathcal{O}(10^5)$ elements). It is also worth recalling that FORTRAN saves matrix elements column-wise. Finally, knowing that the matrix-matrix multiplication is an order $\mathcal{O}(n^3)$ operation, I fitted the curved above with a deg(3) polynomial curve (plotted in dots), which confirms this fact being very well aligned to the runs.
(4) To the best of my knowledge, "gfortran" allows to exploit four major optimization flags while compiling, apart from the basic one: "-O1", "-O2", "-O3", "-Os" (all the documentation can be found here). I collected the computation time for each method with each of these flags, changing the input size by 100 each time in the range $(1; 1000)$. Results are plotted below.







From the plots above it looks clear how the handwritten subroutines can be highly optimized, with any of the basic "gfortran" flags. On the other hand, the built-in method seems to be already optimal (no major changes in CPU time while changing the optimization flag).