Quantum Information and Computing

Academic Year 2024 - 2025

ASSIGNMENT 1

Due for October 28, 2024

Gabriel Amorosetti

gabriel.amorosetti@studenti.unipd.it

1. Setup

- Creation of a working directory on Windows through the terminal

```
PS C:\Users\Gabriel\Documents\Physics of Data\2nd Year\Quantum Information and Computing> mkdir assignment1
```

- Submission of a test job in Fortran :
  - Visual Studio Code 1.94.2
  - Modern Fortran extension 3.2.0
  - GNU Fortran 14.1.0 compiler

```fortran
program Setup
    implicit none   ! To enforce explicit variable declaration
    integer :: a,b
    real :: x, r

    ! Simple calculation as a test job
    x = 3.2
    a = 3
    b = 4
    r = b*x - a

    print *, "Hello world, here is a simple calculation : 4*3.2 - 3 = ", r

end program Setup
```

- Compile with gfortran :

```
gfortran -o a1_ex1.exe a1_ex1.f90
```

- Output :

```
PS C:\Users\Gabriel\Documents\Physics of Data\2nd Year\Quantum Information and Computing\assignment1> .\a1_ex1.exe
Hello world, here is a simple calculation : 4*3.2 - 3 =   9.80000019
```

2. Number precision

- Sum 2.000.000 and 1 with INTEGER*2 and INTEGER*4
  - Implemented in the code with

```fortran
integer(2) ! 16-bit integers
integer(4) ! 32-bit integers
```

- Sum $\pi \cdot 10^{32}$ and $\sqrt{2} \cdot 10^{21}$ with single and double precision
  - Implemented in the code with

```fortran
real(4) ! Single precision real numbers (32-bit)
real(8) ! Double precision real numbers (64-bit)
```

- Compilation error :

```
.\a1_ex2.f90:20:10:

   20 |   a_2 = 2000000   ! Arithmetic overflow, 2 000 000 is too big for a 16-bit integer (only goes from -32 768 to 32 767)
      |          1
Error: Arithmetic overflow converting INTEGER(4) to INTEGER(2) at (1). This check can be disabled with the option '-fno-range-check'
```

- Forcing compilation with '-fno-range-check' and executing :

```
Sum 2 000 000 and 1 with INTEGER*2 :  -31615
Sum 2 000 000 and 1 with INTEGER*4 :   2000001
Sum of pi*10^32 and sqrt(2)*10^21 with single precision :  3.14159278E+32
Sum of pi*10^32 and sqrt(2)*10^21 with double precision : 3.1415926536039354E+032
```

- Limits of 16-bit integers and 32-bit reals :
  - INTEGER*2 can only represent numbers from from -32 768 to 32 767, causing inaccurate results, whereas INTEGER*4 can represent numbers from -2^{31} to 2^{31} - 1, giving the correct sum
  - Single precision ignores the smaller terms with important different magnitudes in the sum, while Double precision takes in account these smaller details, giving a more precise result

3. Matrix multiplication

- 3 approaches to compute A × B = C

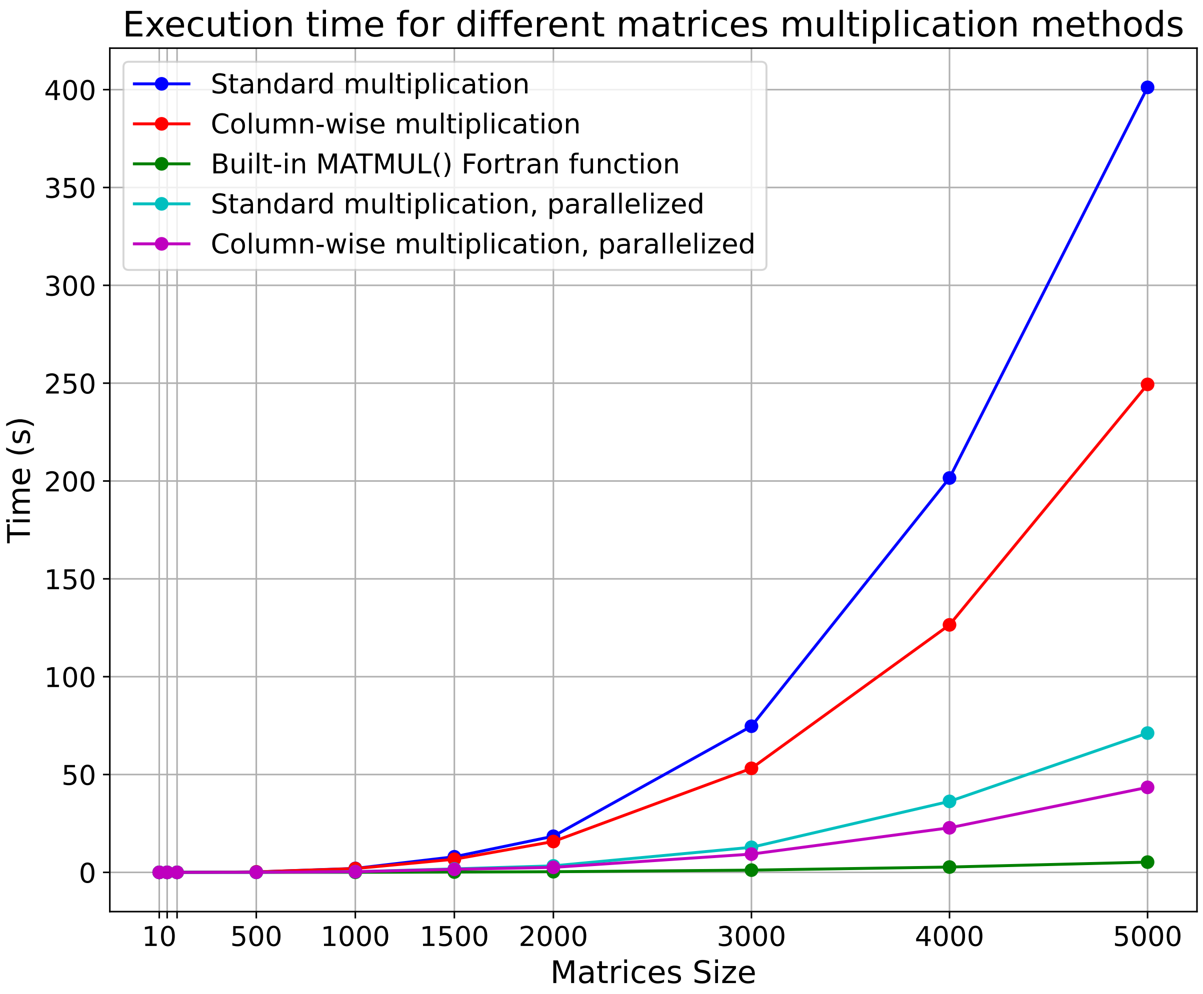
  – Standard multiplication :
  iterates through the rows of the A matrix and the columns of the B matrix,
  filling the output matrix C by row

  – Column-wise multiplication :
  iterates through the columns of the A and B matrices,
  filling the output matrix C by column
    - ➔ Assumed to be faster as Fortran stores arrays in column-major order

  – Fortran intrinsic function :

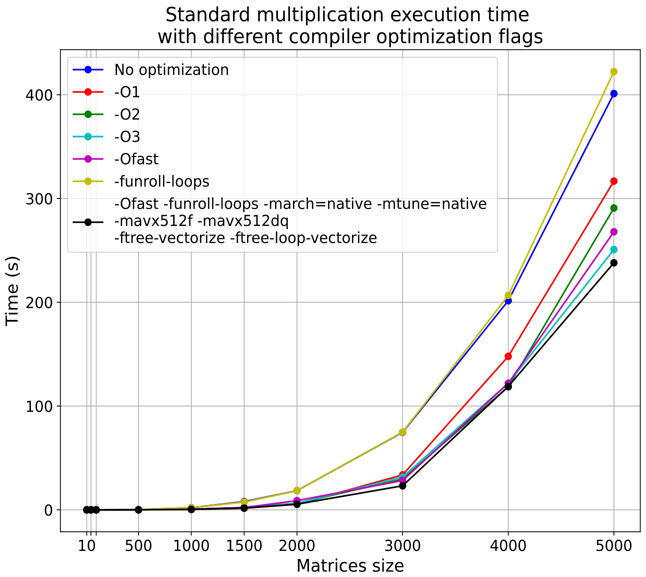
```fortran
C = matmul(A,B)
```

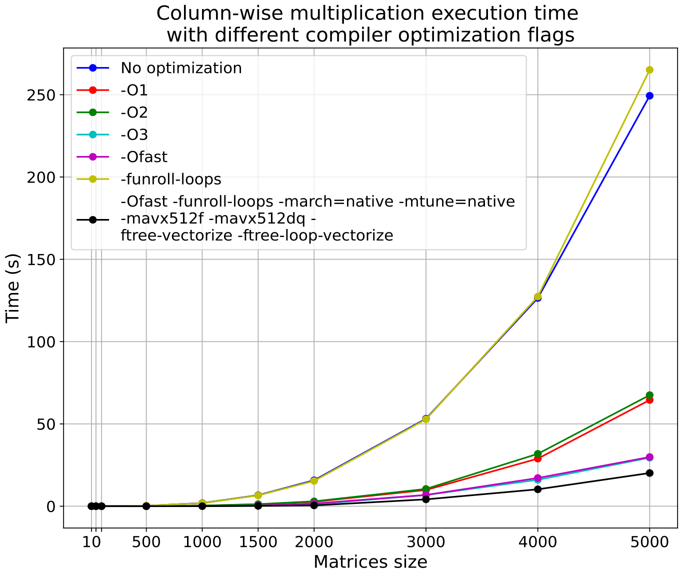
3. Matrix multiplication : methods comparison

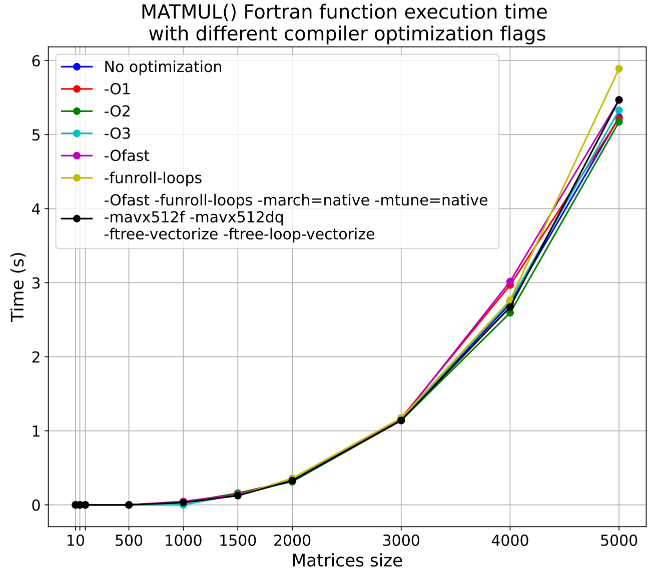


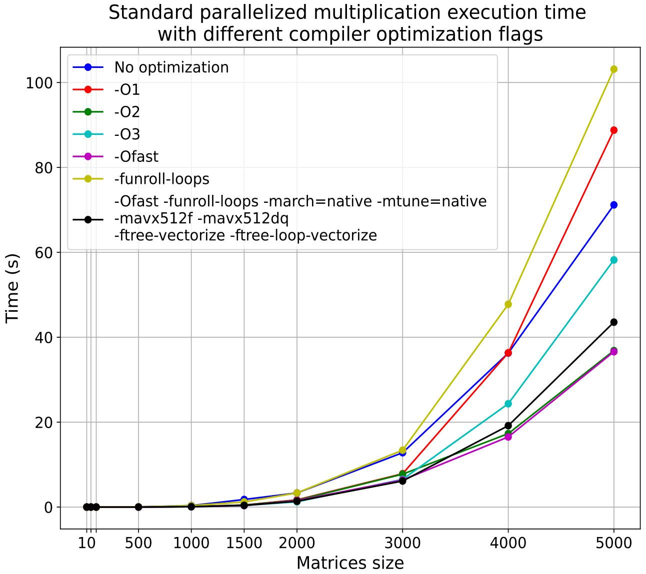
- Additionally, parallelized versions of the standard and column-wise methods were implemented, in order to use all the CPU cores during computation
- Standard multiplication is the slowest one, while the column-wise multiplication performs better, due to a more efficient memory access in Fortran
- Parallelized versions of these methods improve the execution times, the standard parallelized method being still worse than the column-wise one
- The built-in MATMUL() function is significantly faster than the previous approaches, maintaining low execution time for large sizes, as it is the most optimized and efficient

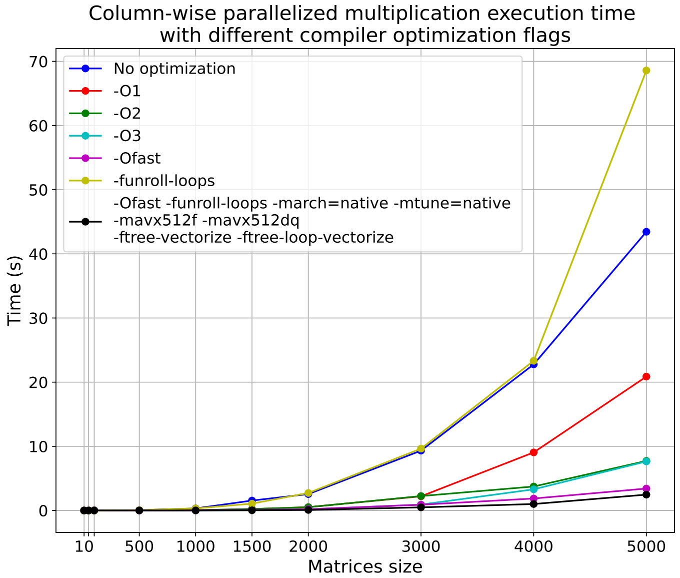
3. Matrix multiplication : optimization flags











- The MATMUL() function does not significantly benefit from any optimization flags, as it already includes low-level optimizations
- The -O flags, listed from the least to most aggressive, improve the execution time according to their respective level of code optimization
- -funroll-loops, performing loop unrolling, actually slows the execution
- The combination of different flags seems to be the most efficient approach here

➔ Further explanations of the different flags are in the code