Simulation and Analysis of 1D Wave Propagation under Various Physical Models

Dario Liotta

Università degli Studi di Padova

Dipartimento di Fisica e Astronomia Galileo Galilei

September 6th 2025

Course of **Quantum Information and Computing**

Academic Year 2024/2025

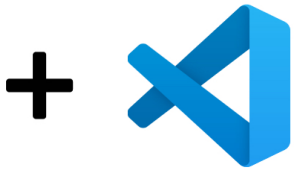
Building up the workspace: the setup

First I created a **working directory**

```
(base) MacBook-Pro-4:Quantum Information and Computing darioliotta$ mkdir Assignments/1
```

The **development environment** I chose

FORTRAN language with
gfortran compiler (v. 14.2.0)



Visual Studio Code with
Modern Fortran extension (v. 3.2.0)

Finally, I submitted the simplest program: `hello_world.f90`

```
1 > hello_world.f90
  1  PROGRAM test
  2      print *, 'Hello World!'
  3  END PROGRAM test
```

Code

```
MacBook-Pro-4:1 darioliotta$ gfortran hello_world.f90 -o hello_world
MacBook-Pro-4:1 darioliotta$ ./hello_world
 Hello World!
MacBook-Pro-4:1 darioliotta$
```

Output

Exploring the limits of INTEGER and REAL in Fortran

Sum of integers

$$2\,000\,000 + 1$$

In Fortran we can select how much space we give to our variables:

- `INTEGER*2`: **2 bytes** $\Rightarrow$ 16 bits $\Rightarrow\ n \in \left[-2^{15}, 2^{15} - 1\right]$
- `INTEGER*4`: **4 bytes** $\Rightarrow$ 32 bits $\Rightarrow\ n \in \left[-2^{31}, 2^{31} - 1\right]$

$2\,000\,000 > 2^{15}$ $\Rightarrow$ **Compiler error**

```
MacBook-Pro-4:1 darioliotta$ gfortran number_precision.f90 -o number_precision
number_precision.f90:9:23:

    9 |     INTEGER*2 :: int2 = 2000000
      |                       1
Error: Arithmetic overflow converting INTEGER(4) to INTEGER(2) at (1). This check
can be disabled with the option '-fno-range-check'
```

Forcing with the `-fno-range-check` flag results in an **overflow**

```
MacBook-Pro-4:1 darioliotta$ gfortran number_precision.f90 -o number_precision -fno-range-check
MacBook-Pro-4:1 darioliotta$ ./number_precision
 Sum of 2,000,000 and 1 with INTEGER*2 (2-byte):       -31615
 Sum of 2,000,000 and 1 with INTEGER*4 (4-byte):      2000001
```

Sum of reals

$$\pi \cdot 10^{32} + \sqrt{2} \cdot 10^{21}$$

Different storage lengths mean different precisions:

- **Single**: **4 bytes** $\Rightarrow$ 32 bits $\Rightarrow$ up to 8 decimal digits
- **Double**: **8 bytes** $\Rightarrow$ 64 bits $\Rightarrow$ up to 16 decimal digits

```
! Suffix _4 and E for exponential are for single precision numbers
print *
print *, 'Pix10^(32) in single precision:', ACOS(-1.0_4) * 1.0E32
print *, 'Pix10^(32) in double precision:', ACOS(-1.0_8) * 1.0D32

! Suffix _8 and D for exponential are for double precision numbers
print *
print *, 'Sqrt(2)x10^(21) in single precision:', SQRT(2.0_4) * 1.0E21
print *, 'Sqrt(2)x10^(21) in double precision:', SQRT(2.0_8) * 1.0D21
```

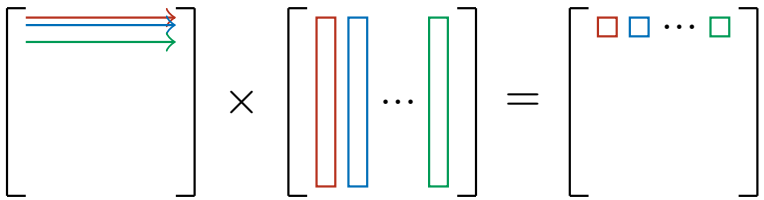
A difference of 11 orders of magnitude means **no difference** in single precision from $\pi \cdot 10^{32}$

```
 Pix10^(32) in single precision:   3.14159278E+32
 Pix10^(32) in double precision:   3.1415926535897933E+032

 Sqrt(2)x10^(21) in single precision:   1.41421360E+21
 Sqrt(2)x10^(21) in double precision:   1.4142135623730950E+021

 Sum with single precision:   3.14159278E+32
 Sum with souble precision:   3.1415926536039354E+032
MacBook-Pro-4:1 darioliotta$
```

Matrix-matrix multiplication: different approaches

Standard approach

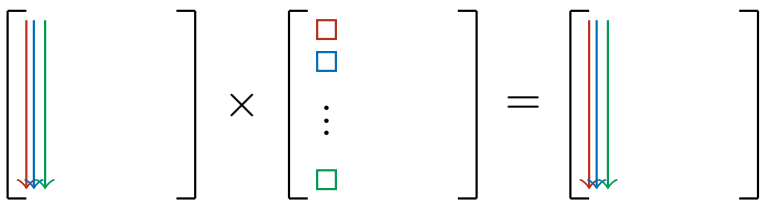
Fill by rows



```fortran
! STANDARD MM MULTIPLICATION FUNCTION
FUNCTION mm_multiplication(A, B, n) RESULT(C)
    IMPLICIT NONE
    REAL, INTENT(IN)    :: A(:,:), B(:,:)  ! Assumed-shape arrays
    INTEGER, INTENT(IN) :: n
    REAL                :: C(n,n)
    INTEGER             :: i, j, k

    C = 0.0   ! Result matrix inizialization

    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

END FUNCTION mm_multiplication
```

Reversed approach

Fill by columns



```fortran
! REVERSED MM MULTIPLICATION FUNCTION
FUNCTION mm_multiplication_columns(A, B, n) RESULT(C)
    IMPLICIT NONE
    REAL, INTENT(IN)    :: A(:,:), B(:,:)  ! Assumed-shape arrays
    INTEGER, INTENT(IN) :: n
    REAL                :: C(n,n)
    INTEGER             :: i, j, k

    C = 0.0   ! Result matrix inizialization

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

END FUNCTION mm_multiplication_columns
```

Fortran approach

Built-in function

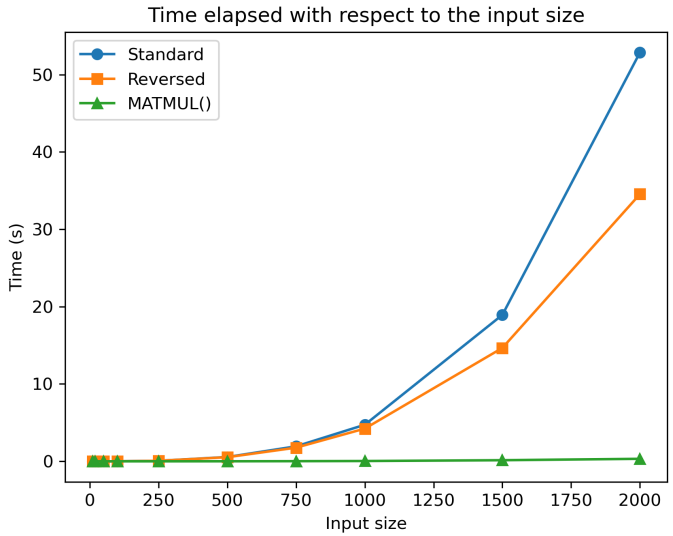
```fortran
REAL ::  A(n,n), B(n,n), C(n,n)  ⟹  C = MATMUL(A,B)
```

Results with different input sizes

- 10 different **input sizes**
- 10 **runs** for each size
- Times evaluated with function `CPU_TIME()`
- Final times are the **average** ones elapsed for each run

```
49    ! RUN FUNCTION
50    FUNCTION run(n) RESULT(t)
51        IMPLICIT NONE
52        INTEGER, INTENT(IN) :: n
53        REAL                :: A(n,n), B(n,n), C(n,n)
54        INTEGER             :: i, j
55        REAL                :: start_time, end_time
56        REAL                :: t(3)
57
58        ! MATRICES INIZIALIZATION
59        do i = 1, n
60            do j = 1, n
61                A(i,j) = real(i+j)
62                B(i,j) = real(i+j)
63            end do
64        end do
65
66        ! STANDARD MATRIX-MATRIX MULTIPLICATION (FILL BY ROWS)
67        CALL CPU_TIME(start_time)
68        C = mm_multiplication(A, B, n)
69        CALL CPU_TIME(end_time)
70        t(1) = end_time - start_time   ! Time estimated
71
72        ! REVERSED MATRIX-MATRIX MULTIPLICATION (FILL BY COLUMNS)
73        CALL CPU_TIME(start_time)
74        C = mm_multiplication_columns(A, B, n)
75        CALL CPU_TIME(end_time)
76        t(2) = end_time - start_time   ! Time estimated
77
78        ! FORTRAN MATRIX-MATRIX MULTIPLICATION
79        CALL CPU_TIME(start_time)
80        C = MATMUL(A, B)
81        CALL CPU_TIME(end_time)
82        t(3) = end_time - start_time   ! Time estimated
83
84    END FUNCTION run
```

Run function

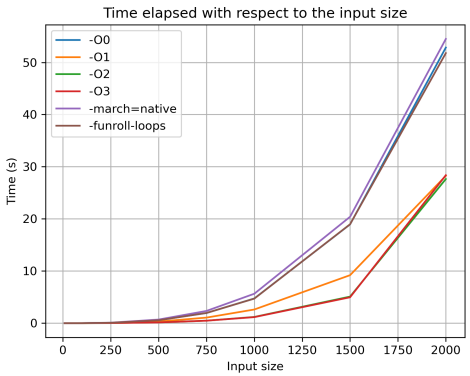


- Standard approach is the worst
- Reversed approach is better, probably because of the **column-major memory layout** (cache optimization)
- `MATMUL()` function is the best, since it's an **intrinsic function**

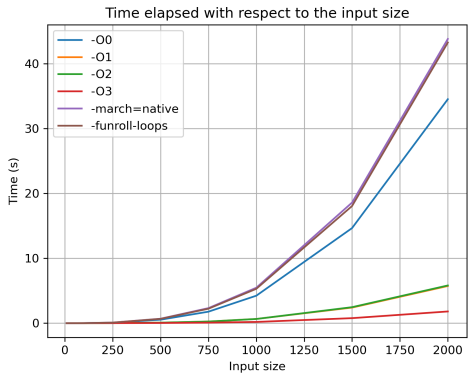
Results with different optimizations

Flag	Description
`-O0`	No optimization
`-O1`	Base optimization; reduces code dimensions and improves performances
`-O2`	Aggressive optimization; enables further options without time compilation increases
`-O3`	Maximum optimization; includes advanced optimization such as unrolling of cycles
`-march=native`	Optimizes code for the CPU architecture where it is compiled
`-funroll-loops`	Unrolls loops to improve performances

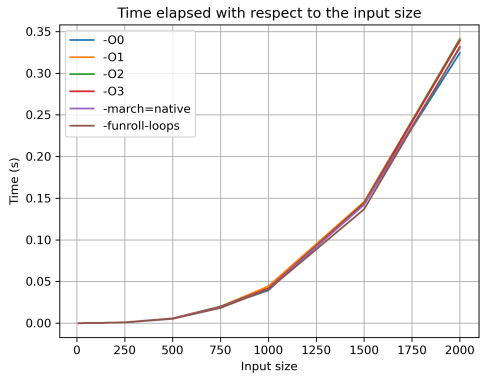
Standard approach



Reversed approach



Fortran approach



- The only difference appeare to be made by `-O1`, `-O2` and `-O3`, both for the **standard** and the **reversed** approach (approximately half of the time)
- All optimization are similar in the **fortran** approach, as it is already much optimized