ASSIGNMENT 1

Physics of Data – Quantum Information and Computing

A.Y. 2024/2025

Lorenzo Calandra Buonaura

24/10/2024

THEORY

- **Number Precision**
  - Computers represent numbers with limited bits, leading to finite precision, which means that, when doing operations, we need to keep in mind that on the borders the computer doesn't work well and could return us unexpected values.
  - In Fortran, INTEGER types store whole numbers with varying sizes: INTEGER*2 stores a 2-byte (16-bit) signed integer, while INTEGER*4 stores a 4-byte (32-bit) signed integer. Thus, when storing a number greater than $2^{16} - 1 = 32{,}767$, we must use INTEGER*4 to avoid overflow, which can occur when trying to store numbers larger than the data type can handle.
  - REAL types represent floating-point numbers and are stored following the IEEE 754 standard, which splits them in three parts (sign, exponent, mantissa). In Fortran we have different precision representation, depending on how many bits are used to store them: single precision is 32 bits, double precision is 64 bits.

- **Matrix-Matrix Multiplication:**
  - Matrix-matrix multiplication is a compute-intensive operation, and optimizing performance requires consideration of algorithm choice, memory management, and hardware capabilities. Standard matrix-matrix multiplication is a P (polynomial) hard problem, with a time complexity of $O(n^3)$ for $n \times n$ matrices.
  - Overall, improving performance in matrix-matrix multiplication relies on optimized algorithms, efficient memory usage, and leveraging parallel hardware to minimize computation time; using different computational options or the intrinsic functions in Fortran can improve the performance a lot.

CODE DEVELOPMENT (1)

- **Test job:**
  - For the test job a simple sub-routine has been used, which has some I/O interaction; the correct functioning of this subroutine ensures that the environment was set correclty.

- **Number precision:**
  - For the integer precision test, we compute the sum 2000000 + 1 twice, first using INTEGER*2 and then INTEGER*4: the compiler already knows that, when using INTEGER*2, it will overflow, thus we need to compile using a flag, '-fno-range-check', which removes range checks and forces compilation.
  - For the real precision test, we compute the sum $\pi * 10^{32} + \sqrt{2} * 10^{21}$ twice, with single and double precision, then we compare the two obtained results.

- **Testing performance:**
  - For the matrix-matrix multiplication we implement three different methods: the first is the standard row by column product, the second is the column by row product and the last leverages Fortran `matmul` function.
  - In order to test the performances we call `cpu_time` before and after each computation, then we subtract the two times to obtained the computational time. We leave the dimension of the matrix `n` as an external input, so that different matrices sizes can be studied with the same executable.
  - Some computation options which allow optimization are employed: -O1 (basic optimizations, improves speed a bit without long compile times), -O2 (more aggressive optimizations for better performance) and -O3 (maximum optimizations). Only the result of –O3 will be showed (most significant).

RESULTS (1)

- The environment is working correctly; the test job runs and produces the expected output.

- The integer precision analysis is in agreement with what expected:
  - For integer*2 we get, forcing the compilation, a value of -31615
  - For integer*4 we get the exact value of 2000001

- The real precision analysis yields the following results:
  - Single Precision Result: 3.14159278E+32
  - Double Precision Result: 3.1415927410267153E+032

  As we can see, when using double precision, we have a different result: we have more precision available, as we can see in the 8th decimal digit, which is now computed accurately. The uncertainty in the double precision case has been moved to the 15th or 16th decimal digit.

- The matrix-matrix multiplication was implemented correctly in the 3 requested ways: reached an upper limit of $n =$ 34235, which is in line with the 10^4 expected limit. After this value, the compiler throws an error, saying that there's no enough memory to allocate the matrices:

```
 Enter the size of the matrices (n x n): 34325
In file 'matmul_timing.f90', around line 34: Error
allocating 4712822500 bytes: Cannot allocate
memory

Error termination. Backtrace:
#0  0x7f7562af9960 in ???
#1  0x7f7562afa4d9 in ???
#2  0x55fdecf090a5 in MAIN__
#3  0x55fdecf0814e in main
```

RESULTS (2)

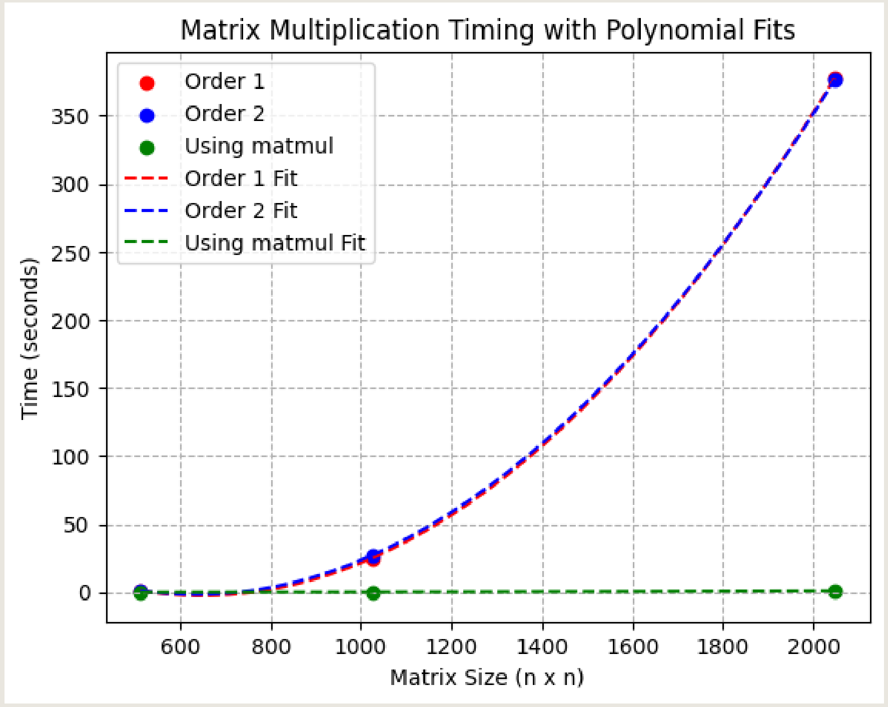
n	Order 1 (s)	Order 2 (s)	Intrinsic `matmul`
512	1.127656	1.176558	0.017480
1024	24.972467	27.027580	0.140167
2048	377.806244	376.491333	1.020264

n	Order 1 (s)	Order 2 (s)	Intrinsic `matmul`
512	0.430553	0.453087	0.018960
1024	8.120380	8.373075	0.108707
2048	102.507225	98.521988	0.666595
4096	833.037476	808.341858	6.123047

- The performance has been tracked averaging over different executions the CPU time. From the upper table (no optimization), we can see that the intrinsic `matmul` function is much more efficient than the row-column and the column-row products implemented.

- The second table reports the CPU times when using optimization (here only –O3, the most significative, is shown): as we can see there is a sensible gain when up-scaling the matrices, even if it doesn't reach the intrinsic function performances.

- The figure shows instead the polynomial behaviour of the compilation times, as expected (the intrinsic function is still polynomial in time, simply much more efficient).


Matrix Multiplication Timing with Polynomial Fits
● Order 1
● Order 2
● Using matmul
Order 1 Fit
Order 2 Fit
Using matmul Fit
Time (seconds)
Matrix Size (n x n)