Scientific Software
Homework 3: Fortran 90, Matrix competition

Emil Løvbak, Wouter Baert & Pieter Appeltans Leuven, November 5, 2020

Introduction

The matrix-matrix multiplication is an important kernel for numerous scientific programs. Hence, an efficient implementation of this operation is a crucial component of any linear algebra library. The goal of this exercise session and the embedded homework is therefore to write a Fortran routine that multiplies two square double precision matrices as fast as possible. To this end, you will have to compare several implementations of the matrix-matrix multiplication and discuss the difference in their efficiency. More specifically, focus on the following concepts:

- compiler independent timings in Fortran;
- optional arguments in Fortran;
- performance comparison between different compilers;
- the effect of optimisation flags (for the gfortran compiler);
- the role of the memory architecture of modern CPUs on the difference in execution time of several implementations of the matrix-matrix multiplication.

During the session

Questions

1. Download the following files from Toledo:
   - `timings.f90`: a module to perform timings in Fortran (similar to `tic` and `toc` in Matlab);
   - `matrixop.f90`: a module that contains the different implementations of the matrix-matrix multiplication;
   - `mm_driver.f90`: a program that performs the timings of the different implementations and that prints a relative error (compared to the result given by `matmul`) for every implementation.
2. Complete the subroutines `tic()` and `toc()` in `timings.f90`. The subroutine `tic()` has an optional argument `startTime`. If this optional argument is not set, the subroutine should record the CPU time at the moment of executing this command in a module variable. Otherwise the CPU time should be returned via the variable `startTime`. The subroutine `toc()` has two optional arguments, `elapsedTime` and `startTime`. If neither argument is present, `toc()` should print the elapsed CPU time since the most recent call of `tic()` (i.e. `tic` called without its output argument). If `elapsedTime` is present, this value should not be printed but returned via `elapsedTime`. If `startTime` is present, the elapsed CPU time since the call of the `tic` command corresponding to `startTime` should be printed/returned.
3. Complete the different implementations of the matrix-matrix multiplication in `matrixop.f90` using the description provided in the following subsection.
4. Compile `mm_driver.f90` with `gfortran -O3 ..  -lblas` and run the resulting executable. Verify your implementations. Compare the execution time of the different methods.

Description different implementations

In this subsection we will use the following notation. Let C be the matrix-matrix product of two $N \times N$ matrices A and B. To denote an element of these matrices we will use a subscript with first the row number and then the column number. For example, the element of A on the row 5 and column 3 is denoted by $A_{5,3}$. Using this notation the element in the i^{th} row and j^{th} column of C is equal to

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j}.$$

The file `mm_driver.f90` contains the following implementations of the matrix-matrix multiplication.

1. `mm_ijk, mm_ikj, mm_jik, mm_jki, mm_kij` and `mm_kji` have three nested loops where the leftmost index changes in the outermost loop, the middle index in the middle loop and the rightmost index in the innermost loop. Example: in `mm_ijk` `i` changes in the outermost loop and `k` in the innermost.
2. `mm_ikj_vect, mm_jki_vect, mm_kij_vect` and `mm_kji_vect` replace the innermost loop by a vector operation such as a scalar-vector multiplication.
3. `mm_ijk_dot_product` and `mm_jik_dot_product` replace the innermost loop by the intrinsic `dot_product` function.
4. `mm_transp_ijk_dot_product, mm_transp_jik_dot_product` are similar to `mm_ijk_dot_product` and `mm_jik_dot_product`, but an additional variable to store the transpose of A is used.
5. `mm_blocks_a` and `mm_blocks_b` use blocking. `mm_blocks_a` is a blocked variant of the fastest method with three nested loops and `mm_blocks_b` is a blocked variant of the slowest method with three nested loops. For sake of simplicity, you may assume that the block size is a divisor of N.
6. `mm_matmul` uses the intrinsic `matmul` function.
7. `mm_blas` calls a BLAS routine to preform the matrix-matrix multiplication.
8. `mm_divide_and_conquer` implements the divide and conquer algorithm for matrix-matrix multiplication.
9. `mm_strassen` implements the Strassen algorithm for matrix-matrix multiplication.

Home work

In this homework we elaborate on the theoretical background of the implementations developed in the exercise session above. Answer the following questions (**Q**), make the necessary figures (**F**), and implement (**I**) the specified functionality.

Remark 1: To easily compare your results, perform the experiments on the PC's in the PC rooms of the department. For questions **F1**, **F2**, **Q9** and **Q10**, specify the machine you used to obtain your answer.

Q1: What is the computational complexity of the straight-forward implementation[1] of the matrix-matrix multiplication of two $N \times N$, double precision floating point matrices? (1 line)

Q2: What is the memory usage of the straight-forward implementation[1] of the matrix-matrix multiplication of two $N \times N$, double precision floating point matrices? (1 line)

I1: Implement the subroutines `startClock()` and `stopClock()` in `timings.f90`. These subroutines have the same functionality as `tic()` and `toc()` from the exercise session, but use wall clock time instead of CPU time and print/return the result in milliseconds instead of seconds.

[1] You can use the provided `mm_ijk` as a reference.

Q3: Compile `mm_driver.f90` with each of the following compilers: gfortran, ifort [2] and nagfor. Use for all compilers the `-O3` optimisation flag. Do you notice a difference between the compilers? (5 lines)

Q4: Compile `mm_driver.f90` with the gfortran compiler and the following compiler flags[3]: `-O0`, `-Og -fbounds-check` and `-O3 -funroll-loops`. Discuss the results. What is the role of the different compiler flags? What do you conclude? (5 lines)

Q5: For gfortran using the optimization flags `-O3 -funroll-loops`, which method with three nested loops is the fastest? (1 line)

Q6: For gfortran using the optimization flags `-O3 -funroll-loops`, which method with three nested loops is the slowest? (1 line)

Q7: Explain the difference in execution time between these two methods. Try to be as detailed as possible. Highlight the important concepts. (10 lines)

I2: Write a Fortran program that prints the size of the matrix and the number of floating point operations per second (in MFLOP/s) for the fastest and slowest method with three nested loops for various matrix sizes. First, increase the matrix size N in steps of 10 for N between 10 and 100. Next, increase N in steps of 100 for N between 100 and 1600. In other words, the matrix size N should take the following values 10, 20, 30, ..., 90, 100, 200, 300, ..., 1600.
Important: To avoid warm-up effects, allocate at the beginning of your program big matrices A, B and C. Compute the matrix-matrix multiplication of A and B using `mm_matmul` and store the result in C. Next, deallocate these matrices and start with your experiment.

F1: Use the output of **I2** (compile with `gfortran -O3 -funroll-loops ...`) to make a figure that shows the number of floating point operations per second in function of N of the fastest and slowest method with three nested loops.

Q8: Briefly discuss Figure **F1**. Avoid repeating information from **Q7**. (8 lines)

Q9: Why would you use blocking? What are good values for the block size given the memory architecture of your machine[4]? **Limit your answer to a theoretical discussion, there is no need to test your actual value.** When using blocking, does it matter which routine you use for the multiplications on block level? Why? (10 lines)

Q10: You can use `valgrind --tool=cachegrind ./executable` with `executable` the name of your executable to simulate the cache architecture of your machine. For the fastest and slowest implementation with three nested loops, compare the memory efficiency of the variant without blocking and the blocked variant for blocksizes 25, 100 and 500. Also compare the execution time of these variants (run them without `valgrind`). Write several short programs to perform this experiment and compile them with `gfortran -O3 ...`. Use N equal to 2000 for the fastest method and N equal to 1500 for the slowest method. Discuss your results. (7 lines)

F2: As in **F1**, plot the number of floating point operations per second in function of N for `mm_blocks_a` and N between 100 and 1600 using a blocksize of 100.

Q11: Briefly discuss the difference between **F1** and **F2**. (2 lines)

Q12: Discuss briefly the practical relevance of the divide and conquer algorithm. A reference implementation - which assumes $N = 2^k$- is provided in `matrixop.f90`. (3 lines)

Q13: Discuss briefly the practical relevance of the Strassen's algorithm. A reference implementation - which assumes $N = 2^k$- is provided in `matrixop.f90`. (3 lines)

[2] If you get a segmentation fault, adding the flag `-heap-arrays 0` might resolve the problem.

[3] More information on the compiler options that control the optimization level can be found on `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

[4] you can use `lscpu | grep cache` or `getconf -a | grep CACHE` to inspect the cache sizes of your machine

I3: Download `blas_divide_and_conquer.f90`, which does one step of the divide and conquer algorithm and then uses BLAS to multiply the submatrices, from Toledo. Complete this file: choose the appropriate BLAS routine and complete the subroutine calls. For simplicity, you may assume that N is divisible by 2.

Q14: How can you verify if a call of `mm_blas` uses multiple cores? (2 lines)

Practical information

This is a smaller homework, so try to spend no more than 12 hours on this homework (excluding time spend on the self study). You should submit a zip containing the following files on Toledo before **16 November at 14h00**. Your zip should have as name `hw2_lastname_firstname_studentnumber.zip` with `lastname` your last name, `firstname` your first name and `studentnumber` your student number. For example if your name is John Smith and your student number is r0123456, your file should be called `hw2_smith_john_r0123456.zip`.

- The Fortran code for **I1**, **I2** and **I3** and `matrixop.f90`. Give at the start of your code the commands you used to compile and link the files. When there are more then a few instructions, use a Makefile.
- A PDF with the desired figure(s) and answers. Use the template provided on Toledo. Try to be concise, there is no need for full sentences; telegram style writing suffices. Try to cover the bullet points from the introduction and the material from the **self study** in your answer. Make sure your submission is inline with the introductory guidelines.