Technisch Wetenschappelijke Software
Scientific Software
Session 5 & Homework 3:
Matrix-matrix multiplication

Introduction

The goal of this exercise session and the embedded homework is to examine several implementations of the matrix-matrix multiplication in Fortran and discuss the difference in their execution time. More specifically, we will 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 computers 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, which you will have to complete in step 3;
   - `mm_driver.f90`: a program that performs some basic 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 `real(kind(0e0))` 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 the `real(kind(0e0))` variable `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 descriptions provided in the following subsection.
4. Compile `matrixop.f90`, `timings.f90` and `mm_driver.f90` with `gfortran -O3`. Link the resulting object files (do not forget `-lblas`). Finally, run the resulting executable. Verify your implementations. Compare the execution times of the different variants.

Descriptions of the different implementations

In this subsection we will use the following notation. The $N \times N$ dimensional matrix C is the matrix-matrix product of two $N \times N$ dimensional matrices A and B,

$$C = A * B.$$

To denote an element of these matrices we will use a subscript with first the row index and then the column index. For example, the element of A on the fifth row and third column is denoted by $A_{5,3}$. Using this notation the element on 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 `matrixop.f90` contains a skeleton for 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. For example, in `mm_ijk` `i` changes in the outermost loop, `j` in the middle loop and `k` in the innermost loop.

2. `mm_ikj_vect, mm_jki_vect, mm_kij_vect` and `mm_kji_vect` replace the innermost loop by a scalar-vector multiplication, e.g., for `ikj` you have to replace the inner most loop by `C(i,:) = C(i,:) + A(i,k)*B(k,:)`.

3. `mm_ijk_dot_product` and `mm_jik_dot_product` replace the innermost loop by the `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 use an additional variable to store the transpose of A.

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. To implement these methods, **use six nested loops.** The order of the outer and inner loops should be the same. For example, if you make a blocked version of `mm_kij` then your loops (from outermost to innermost loop) should iterate over k-i-j-k-i-j.

6. `mm_matmul` uses the intrinsic `matmul` function.

7. `mm_blas` calls a BLAS routine to preform the matrix-matrix multiplication[1].

8. `mm_divide_and_conquer` implements the divide-and-conquer algorithm for the matrix-matrix multiplication.

9. `mm_strassen` implements the Strassen algorithm for the matrix-matrix multiplication.

Homework

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 be able to easily compare your results, we ask you to perform your timings on one of the PCs in the departmental PC rooms. For questions **F1**, **F2**, **Q9** and **Q10**, specify the machine you used to obtain your answer.

Important: When performing timings, make sure that there are no other users on the machine you use. Use `htop` to monitor the active processes.

[1] BLAS uses multiple CPU cores to perform the matrix-matrix multiplication. So, the CPU time will not give correct results on the real execution time in this case. But, for this exercise session this difference can be ignored.

Q1: What is the computational cost of the straight-forward implementation[2] of the matrix-matrix multiplication of two $N \times N$, double precision floating point matrices?

Q2: What are the storage requirements of the straight-forward implementation[1] of the matrix-matrix multiplication of two $N \times N$, double precision floating point matrices?

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.

Q3: Compile `matrixop.f90`, `timings.f90` and `mm_driver.f90` with each of the following compilers: `gfortran`, `ifort` [3] and `nagfor`. For each compiler, use the `-O3` optimisation level. Discuss the most important differences between the compilers.

Q4: Compile `matrixop.f90`, `timings.f90` and `mm_driver.f90` with the `gfortran` compiler and the following compiler flag combinations[4]: `-O0`, `-Og -fbounds-check`, `-O3 -funroll-loops` and `-Ofast -march=native -ffast-math`[5]. Compare the results. What is the role of the different compiler flags? What do you conclude? When would you use which combination?

Q5: For `gfortran` using the optimization flag `-O3`, which method with three nested loops is the fastest?

Q6: For `gfortran` using the optimization flag `-O3`, which method with three nested loops is the slowest?

Q7: Explain the difference in execution time between these two methods. Try to be as detailed as necessary, but remain to the point. Highlight the most important concepts using **boldface**.

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. Your output should look as follows:

```
10 1500.32534 400.234234
20 2304.62839 260.138745
...
```

Important: To avoid warm-up effects, allocate at the beginning of your program sufficiently large (e.g., $N = 2000$) matrices A, B and C. Then, compute the matrix-matrix multiplication of A and B using `matmul` and store the result in C. Next, deallocate these matrices and start with your experiment. Also, for each matrix size, initialize A and B with random values. A starting point file, named `i2.f90` is provided on Toledo.

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

Q8: Briefly discuss Figure **F1**. Avoid repeating information from **Q7**.

Q9: What are the (dis)advantages of blocking?

Q10: You can use `valgrind --tool=cachegrind ./executable` with `executable` the name of your executable to simulate the cache architecture of your machine[6] for the given executable. For the fastest and slowest implementation with three nested loops, compare the cache efficiency of the

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

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

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

[5] If you get a segmentation fault, try adding the flag `-fmax-stack-var-size=0`.

[6] To inspect the size of the caches of your machine, you can run `getconf -a | grep CACHE`.

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. Do you get the results you expect based on the previous questions? Discuss.

F2: As in **F1**, plot the number of floating point operations per second in function of N for `mm_blocks_a` and `mm_blocks_b`. Do this for N between 100 and 1600 (steps of size 100) and a blocksize of 100.

Q11: Briefly discuss the difference between **F1** and **F2**.

Q12: Briefly discuss the practical relevance of the divide and conquer algorithm. A reference implementation - which assumes $N = 2^k$- is provided in `matrixop.f90`.

Q13: Briefly discuss the practical relevance of the Strassen's algorithm. A reference implementation - which assumes $N = 2^k$- is provided in `matrixop.f90`.

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.

Practical information

This is a smaller homework, so try to spend no more than 15 hours on this homework (excluding time spend on the self study). You should submit a zip with your results on Toledo before **Wednesday, November 29th at 14h00**. This zip should be named `hw3_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 `hw3_smith_john_r0123456.zip`. Your zip should contain the following files:

- The Fortran code for **I1** (in `timings.f90`), **I2** (in `i2.f90`) and **I3** (in `blas_divide_and_conquer.f90`) and `matrixop.f90`. Only submit source files, do not include `.o` or `.mod` files and executables in your zip.
- A PDF with the desired figure(s) and answers. Use the template `hw3_lastname_firstname_studentnumber.tex` provided on Toledo. Try to be complete but remain to the point, 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.
- A text file named `time_spent.txt`, only containing a single number, the amount of hours you spent on this assignment. This has no influence on your grade but helps us in determining the average workload of each assignment for future years, so please be honest.

As before, if you take the Dutch course, you can write your code and report in Dutch. Finally, make sure your submission contains the necessary files and is inline with the introductory guidelines.