Optimization and OpenMP parallelization of addition and product computations of two dense matrices.

HPC 4MA 2025/2026

F. Couderc and L. Giraud

Abstract

We will focus in this work on:

- the optimization of the addition computation of two dense matrices by improving memory accesses,
- the OpenMP parallelization of the dense matrix-matrix product computation using the maximum resources available on your machine to distribute loops across the available cores, as an optimization by improving memory accesses by the so called cache blocking principle.

A report is required on this practical work: short and as precise as possible with generation of graphics.

Program

A **`blas.c`** program is provided that contains:

- a **`int main`** function with, after some initializations, contains a loop calling at each iteration the addition of two matrices or the product of two matrices using different approaches (one is chosen at compilation, see compilation section below), both several times (`ncase` variable), growing the problem size from iteration to other (see the predefined variables at the beginning of the program in order to control it). At each iteration, the mean time value and the mean GFlops/s are reported in a file ***`timings.txt`*** (at the root directory of the program, in csv format) in addition to terminal screen outputs:
  - BLAS1 routines are vector-vector operations like the dot product calling the **`cblas_ddot`** routine and one need then to call the routine for each entry of the resulting matrix in order to perform the entire matrix-matrix product.
  - BLAS2 routines are matrix-vector operations like the matrix-vector product calling the **`cblas_dgev`** routine and then one need to call the routine for each column of the resulting matrix in order to perform the entire matrix-matrix product.
  - BLAS3 routines are matrix-matrix operations and one call to **`cblas_dgemm`** perform the entire matrix-matrix product.
- **`void init_matrix(double *A, int ld, int nrow, int ncol, double cst)`**: a function to initialize the entries of a matrix according to the formula $A_{ij} = \text{cst}/\sqrt{\text{nrow}}/\sqrt{\text{ncol}}$. The matrix A has **`nrow`** rows and **`ncol`** columns stored in a column-major fashion with a leading dimension **`ld`**.
- **`double norm_matrix(double* A, int nrow, int ncol, int ld)`**: a function returning the computation of the Frobenius norm of the matrix A.
- **`void print_matrix(double* A, int nrow, int ncol, int ld)`**: a function to print on the terminal screen the matrix A in order to check the correctness of the calculation for matrices of small sizes.

- **`inhouse_add(double* A, double* B, double* C, int N, int ld)`**: a in-house version of the addition of a matrix A with a matrix B, returning the result in a matrix C, all of size $N \times N$.
- **`inhouse_dot(double* A, double* B, double* C, int N, int ld)`**: a in-house version of the product of a matrix A by a matrix B, returning the result in a matrix C, all of size $N \times N$.

The matrices entries are stored using one-dimensional arrays linking a_{ij} to `A[i+ld*j]`. The matrix representation is then in column-major order since the consecutive elements of a column reside next to each other contiguously in memory, which is important to understand to optimize memory accesses[1].

We recall that the performance is commonly measured in GFlops/s and the matrices product needed $2N^3$ Flops to be computed.

Compilation linking the OpenMP and BLAS libraries

The complete compiler directive to be written in a terminal window (where the program is placed) is,

compilation directive

```
gcc -o blas blas.c -O3 -lblas -fopenmp -lm {-DUSE_ADD, -DUSE_MUL, -DUSE_BLAS1, -DUSE_BLAS2
    or -DUSE_BLAS3}
```

where,

- **`gcc`** is the GNU free C compilator and **`-o blas`** is the option to redirect the produced binary executable to the file name blas (`./blas` to execute it),
- **`-O3`** is for activating the ("aggressive") optimization in the compilation, strictly needed for coherent computing time results,
- **`-lblas`** is for searching and linking the BLAS library,
- **`-fopenmp`** is for searching and linking the OpenMP library,
- **`-lm`** is for searching and linking the `math.h` library (needed to compute sqrt),
- **`{-DUSE_ADD, -DUSE_MUL, -DUSE_BLAS1, -DUSE_BLAS2 or -DUSE_BLAS3}`** is needed and allow to call, by the use of preprocessor directives (`if`, `elif` and `endif`), the corresponding piece of source code.

The BLAS library which is very optimized and parallelized is used to check all the supercomputer peak performance (called the LINPACK benchmark [2]), as it really uses all processors resources to perform the computation. It will be used here as the reference time computation to study the difference with our home-made implementations.

Modify the memory accesses to add two matrices

Rewrite the **`inhouse_add_reorder`** function changing the loop order to enhance the memory accesses by ensuring that the new function complies with the spatial locality property.

[1]Note that in the C programming language that `a[i][j]` is in row-major oder: Wiki.
[2]https://www.top500.org/project/linpack/

- Check the performances between the two versions and report the results in a graphical way.
- Explain briefly why the cache accesses are better in this case (to be seen with the instructor).

Test the BLAS implementations

- Test the BLAS{1,2,3} implementations of the matrix-matrix implementations.
- Check the performances and report the results in a graphical way.

OpenMP parallelization

Now, the `inhouse_dot` function need to be parallelized (as the `init_matrix` and `norm_matrix`). The selected loop will be distributed across threads using the `runtime` scheduling in order to control it by the environment variable `OMP_SCHEDULE` (for example, by typing the command `export OMP_SCHEDULE=dynamic,8` in the terminal before calling the program, meaning that the scheduling policy will be `dynamic` with a chunk of `8` for all loops involved in the multi-threaded processes you will next run from this terminal window).

- Control eventually the correctness of the parallelization on small matrices (for example $N = 8$) by printing to screen the computed matrix C.
- Check again the performances and report the results in a graphical way.
- Also check if the scheduling mode as the chunk size can improve the performances.

Use cache blocking

While significant results should have been already obtained by parallelizing the computation, we can do even better regarding cache locality. This can be achieved by constructing matrix-matrix product sub-problems of much smaller size than the initial one based on the programming model:

cache blocking kernel

```
for (j=0; j<M; j+=BLOCK)
   for (k=0; j<K; k+=BLOCK)
      for (i=0; i<N; i+=BLOCK)
         for (jj=0; jj<BLOCK; jj++)
            for (kk=0; kk<BLOCK; kk++)
               for (ii=0; ii<BLOCK; ii++)
```

- Finish to write the function `inhouse_dot_blocking` with the cache blocking principle and with again the OpenMP pragma to distribute the computation across threads.
- Check again the performances obtained and report the results in a graphical way.
- Explain briefly why the cache accesses are better in this case (to be seen with the instructor).