Optimization and OpenMP parallelization of the dense matrix-matrix product computation.

HPC 4GMM 2021/2022

F. Couderc and L. Giraud

Abstract

We will focus in this work both on the optimization and OpenMP parallelization of the dense matrix-matrix product computation in order to reduce the computation time by using the maximum of available resources on your machine. We will see how it is possible to reduce the computation time by using OpenMP to distribute computation loops across the avaible cores as well as by enhancing the memory accesses.

Introduction

A `blas3.c` program is provided that contains:

- `void init(int nrow, int ncol, int ld, double* A, double c)`: a function to initialize the entries of a matrix A according to the formula $a_{ij} = c\,(i+j)/\text{nrow}/\text{ncol}$. The matrix A has `nrow` rows and `ncol` columns stored in a column-major fashion with a leading dimension `ld`.
- `double norm(int nrow, int ncol, int ld, double* A)`: a function returning the computation of the Frobenius norm of a matrix A.
- `void print_array(int ld, int nrow, int ncol, double* A)`: a function to print on the screen the matrix A in order to check the correctness of the calculation for matrices of small sizes.
- `void naive_dot(double *A, int lda, double *B, int ldb, double *C, int ldc)`: a function of the so-called naive version of the matrices product of a matrix A of size $N \times K$ by a matrix B of size $K \times M$, returning the result in a matrix C of size $N \times M$.
- and of course a `main` with some initializations and some performance measurements of the matrix-matrix product using the naive implementation and the optimized BLAS implementation available on your computer (`cblas_dgemm` function).

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 $2NKM$ Flops to be computed.

[1]Note that in the C programming language that `a[i][j]` is in row-major oder: Wiki.

Compilation linking the OpenMP and BLAS libraries

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

```
gcc -o blas3 blas3.c -O3 -lblas -fopenmp -lm
```

where,

- `gcc` is the GNU free C compilator and `-o blas3` is the option to redirect the produced binary executable to blas3 (`./blas3` to execute it),
- `-O3` is for activating the (agressive) optimization in the compilation, strickly needed for coherent computing time results,
- `-lblas` is for searching and linking the BLAS library,
- `-fopenmp` if for searching and linking the OpenMP library,
- `-lm` is for searching and linking the `math.h` library (needed to compute sqrt).

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 ressources 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

Rewrite the **`saxpy_dot`** function changing the loop order according to the original **`naive_dot`** function in order to enhance the memory accesses by ensuring that the new function complies with the spatial locality property.

Check the performances obtained using large matrix sizes (for example $N = M = K = 2048$) and report the results.

OpenMP Parallelization

Now, the **`init`**, **`norm`**, **`naive_dot`** and **`saxpy_dot`** functions need to be parallelized. 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 chunck of **`8`** for all loops involved in the multi-threaded processes you will next run from this terminal window).

Control the correctness of the parallelization on small matrices (for example $N = M = 4$ and $K = 8$) by printing to screen the computed matrix C.

Check again the performances obtained using large matrix sizes (for example $N = M = K = 2048$) and report the results.

[2]https://www.top500.org/project/linpack/

Use cache blocking

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

cache blocking kernel

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

Rewrite the **`blocking_dot`** function with the good loop order, the cache blocking principle and with again the OpenMP pragma to distribute the computation across threads.

Check again the performances obtained using large matrix sizes (for example $N = M = K = 2048$) and report the results.

Explain why the cache accesses are better in this case.