Rapport de Travaux Pratiques

Domain Decomposition and iterative Solvers

Mention : dense matrix product and BLAS
Filière : Lab 1
Computaional Mechanics

14/09/2023

Tuteurs école :
Nicolas Chevaugeon
Nicolas.Chevaugeon@ec-nantes.fr
`https://cv.hal.science/nicolas-chevaugeon`

Etudiant :
Li Zichen
Zichen.Li@eleves.ec-nantes.fr

Résumé

In this project, we delve into the computational realm of dense matrix multiplication, a cornerstone operation in numerous scientific and engineering applications. Utilizing C++ as the primary programming language, we harness the power of the Basic Linear Algebra Subprograms (BLAS) library, particularly the dgemm function, to perform these matrix operations. A critical facet of our investigation lies in contrasting the efficiencies of matrix multiplication achieved through different for loop orders. This comparative analysis provides insights into the optimal loop arrangements for maximizing computational speed. To validate the accuracy and efficiency of our implementations, we employed Python scripts. These scripts not only served as a verification tool but also facilitated in plotting the relationship between matrix sizes and computational efficiencies across various multiplication techniques. Furthermore, we venture into the domain of parallel computing, discussing its potential to significantly enhance the speed and efficiency of matrix operations. Through this comprehensive study, we aim to shed light on best practices and potential avenues for optimizing matrix multiplication processes in real-world applications.

In the submitted assignment files, we provide three versions of 'main.cpp'. They correspond to verifying the multiplication results ('main_check.cpp'), selecting the best two loop orders ('main_choose.cpp'), and plotting the computation speed graph ('main_figure.cpp'). Before running, please remember to rename the desired file back to 'main.cpp'.

Table des matières

1 Introduction and set-up

1.1 Definition

Given two matrices A of size $m \times p$ and B of size $p \times n$, their matrix product C will be a matrix of size $m \times n$. The entry in the i^{th} row and j^{th} column of matrix C, denoted as $C(i,j)$, is computed as the dot product of the i^{th} row of matrix A and the j^{th} column of matrix B.

Mathematically, this can be represented as :

$$C(i,j) = \sum_{k=1}^{p} A(i,k) \times B(k,j)$$

for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

In this project, we will analyze the impact of different **for**loop orders on code performance. To simplify the description, taking the above figure as an example, such a loop will be referred to as the "mnp" loop.

1.2 Algorithm

A basic algorithm to compute the matrix multiplication can be described using nested loops :

```
for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
        for $k = 1$ to $p$ do
            C(i,j) = A(i,k)*B(k,j) +C(i,j)
        end
    end
end
```

1.3 Complexity

The time complexity of this basic algorithm is $O(m \times n \times p)$. For square matrices of size $n \times n$, this becomes $O(n^3)$.

While this basic algorithm is straightforward, it may not be the most efficient for all applications, especially for larger matrices. Various optimization techniques, such as parallel computing or the use of BLAS libraries, can improve efficiency. Additionally, the storage pattern of the matrices (row-major or column-major) and the order of the loops can impact cache performance and thereby affect the computation speed.

In our example, matrix A is a matrix with M rows and K columns, while matrix B has K rows and N columns. The computed matrix C will have M rows and N columns. Therefore, we will test six loop orders : "NKM", "KNM", "NMK", "KMN", "MKN", and "MNK". We will perform parallel computations on the most efficient loop and ultimately compare it with the 'dgemm " function in the BLAS library.

2 Methods

2.1 Implementation of matrix multiplication

We will first consider the dmatix_denseCM.cpp file and begin by implementing the loop.

```cpp
/* NKM */
dmatrix_denseCM mulV1(const dmatrix_denseCM &A, const
    dmatrix_denseCM &B){
   const size_t M = A.getNbLines();
   const size_t K = A.getNbColumns();
   const size_t N = B.getNbColumns();
   if (K != B.getNbLines()){
       std::cout <<"error can't compute product of 2 matrices :
           dimension missmatch." << std::endl;
        throw;
   }
   dmatrix_denseCM C(M,N,0.);
   const double *a = A.data();
   const double *b = B.data();
   double       *c = C.data();
   for (size_t k = 0; k < N; ++k)
   {
      for (size_t j = 0; j < K; ++j)
      {
          for (size_t i = 0; i < M; ++i)
          {
              *(c + k * M + i) += (*(a + j * M +i)) * (*(b + k*K
                  + j));
          }
      }
   }
```

Similarly, we implement the other five loop orders : "KNM", "NMK", "KMN", "MKN", and "MNK". Using these loop orders, we then create objects of the dmatrix_denseCM class named mulV2, mulV4, mulV5, mulV6, and mulV7 respectively.

2.2 Multiplication answer verfication

To verify our matrix multiplication results, we first need to set matrices A and B as rectangular matrices in the 'main.cpp' file. Then, we output the matrix results to the 'output.txt' document. Following that, we will write a Python script, check_result.py, and leverage the existing 'numpy' library to validate the calculations.

```
/* main.cpp */
  const size_t n = 4;
  const size_t k = 3;
  const size_t m = 6;

  mat A(n,k);
  mat B(k,m);
```

Here, a matrix A with four rows and three columns is generated, along with a matrix B that has three rows and six columns.

Using the code below, we can open and close the file at any location within the code, allowing us to write to the same file multiple times. However, if we don't open the file using the std : :ios : :app mode, then every time you open the file, the original content will be overwritten. Here, we want to append content in different mulV1, mulV2, mulV3, muldgemm, mulV4, mulV5, mulV6, and mulV7 instead of overwriting. We can open the file as follows :

```
/* main.cpp */
#include <fstream>
    //std::ofstream outfile("output.txt", std::ios::app);
    std::ofstream outfile1("output.txt");
    if (outfile1.is_open()) {
         outfile1 << A;
             outfile1.close();
    } else {
            std::cerr << "Unable to open file for writing." <<
                std::endl;
    }
```

Opening and closing files frequently can impact code performance. File I/O (Input/Output) operations, especially opening and closing files, are quite costly compared to memory operations. This effect is particularly evident when dealing with large amounts of data or frequent writes. Each opening and closing of a file involves interaction with the operating system, leading to the following overheads :

- System call overhead : File operations require system calls, which necessitate switching from user mode to kernel mode.
- File locking : Opening a file may involve the acquisition and release of file locks.

- Disk I/O : Frequent file I/O can lead to disk operations, which are relatively slow.
- File buffer flushing : Closing or writing to a file might require flushing buffers to disk, a high-cost operation.

Here, after creating and opening 'outfile1', we will close 'outfile1' after 'mulv7'.

Next, we will use a Python script to read the output file "output.txt", invoke the matrix dot product(Resulting Matrix) from the numpy library using numpy.dot, and compare it with our results.

```
#check_result.py
import numpy as np
def read_matrices_from_txt(file_path):
    matrices = []
    with open(file_path, 'r') as file:
        while True:
            rows, cols = map(int, file.readline().split())
            matrix = []
            for _ in range(rows):
                row = list(map(float, file.readline().split())
                    )
                matrix.append(row)
            matrices.append(matrix)
            # Check if we have reached the end of the file
            pos = file.tell()
            if file.readline() == '':
                break
            file.seek(pos)  # Reset position if not at end

    return matrices

file_path = "/home/zli2022/DDIS/lab1-test/output.txt"
matrices = read_matrices_from_txt(file_path)
for matrix in matrices:
    for row in matrix:
        print(row)
    print("-----")

result_matrix = np.dot(matrices[0], matrices[1])

print("Resulting Matrix:")
print(result_matrix)
print("-----")

# Compare the result with the other matrices
for i, matrix in enumerate(matrices[2:], 3):  # Start index
```

```
        from 3 for printing purposes
    is_same = np.allclose(result_matrix, matrix)  # Compares
        the matrices with some tolerance
    print(f"Matrix {i} is {'the same as' if is_same else '
        different from'} the resulting matrix.")
```

Here we display a portion of the output. In which, Matrix3, 4, 7, 8, 9, and 10 respectively represent the results of the "NKM", "KNM", "NMK", "KMN", "MKN", and "MNK" loop orders. Matrix5 represents the result of the "NKM" computation implemented using OpenMP, while Matrix6 represents the result of the dgemm function computation.

```
Resulting Matrix:
[[0.79665758 0.39833516 0.62808414 0.60683309 0.84663219
   0.58381018]
 [0.51622291 0.25402844 0.53677292 0.38094274 0.70697586
   0.42654678]
 [1.37338116 0.85598206 0.94298341 1.13605955 1.37175362
   0.99804701]
 [1.25379284 0.72752696 0.82086018 1.01525212 1.17577095
   0.881687  ]]
-----
Matrix 3 is the same as the resulting matrix.
Matrix 4 is the same as the resulting matrix.
Matrix 5 is the same as the resulting matrix.
Matrix 6 is the same as the resulting matrix.
Matrix 7 is the same as the resulting matrix.
Matrix 8 is the same as the resulting matrix.
Matrix 9 is the same as the resulting matrix.
Matrix 10 is the same as the resulting matrix.
```

Following the above process, we can confirm that our matrix multiplication implementation is correct. Next, we will determine which two loop orders are the most efficient and implement them using OpenMP. We will then compare their arithmetics speed with that of the 'dgemm' function.

2.3 Arithmetics speed

We primarily conduct file writing and multiplication operations through the dmatrix_denseCM class within the 'main.cpp' file. Given the significantly faster computation speeds achieved using the dgemm function and OpenMP optimization compared to regular for loops, we choose the fastest among the six for loops to implement with OpenMP acceleration. Then, we compare the computation speeds among this OpenMP-accelerated version, two fastest loop, and the dgemm function implementation.

We will perform matrix multiplication on square matrices. To compare the speeds of different for loops, we choose a larger size of 1000x1000. At the same time, we set the boolean variable verbose to false, so that all the elements of the matrix won't appear in the terminal window.

```cpp
/* main.cpp */
int main(){
  bool verbose = false;
  const size_t n = 1000;
  mat A(n,n);
  mat B(n,n);
```

Below are the results from the terminal :

```
1000x1000 prod computed in time_mult 0.277802s for NKM
 Gflop/s 7.19938
1000x1000 prod computed in time_mult 0.305217s for KNM
 Gflop/s 6.55271
1000x1000 prod computed in time_mult 0.052793s for NKM with
   OpenMP
 Gflop/s 37.8838
1000x1000 prod computed in time_mult 0.0109781s for dgemm
 Gflop/s 182.18
1000x1000 prod computed in time_mult 1.07654s for NMK
 Gflop/s 1.8578
1000x1000 prod computed in time_mult 7.12819s for KMN
 Gflop/s 0.280576
1000x1000 prod computed in time_mult 7.73177s for MKN
 Gflop/s 0.258673
1000x1000 prod computed in time_mult 7.97766s for MNK
 Gflop/s 0.2507
```

From our comparison, we determined that "NKM" and "KNM" are the two fastest for loop orders for computation, with "NKM" being the fastest among the six loop orders. Therefore, we chose "NKM" for parallelization with OpenMP in mulV3.

```cpp
/* dmatrix_denseCM.cpp */
#include <omp.h>
dmatrix_denseCM mulV3(const dmatrix_denseCM &A, const
   dmatrix_denseCM &B){
  const size_t M = A.getNbLines();
  const size_t K = A.getNbColumns();
  const size_t N = B.getNbColumns();
  if (K != B.getNbLines()){
      std::cout <<"error can't compute product of 2 matrices :
          dimension missmatch." << std::endl;
```

```
        throw;
    }
    dmatrix_denseCM C(M,N,0.);
    const double *a = A.data();
    const double *b = B.data();
    double       *c = C.data();

    #pragma omp parallel for
    for (size_t k = 0; k < N; ++k)
    {
      for (size_t j = 0; j < K; ++j)
      {
          for (size_t i = 0; i < M; ++i)
          {
              *(c + k * M + i) += (*(a + j * M +i)) * (*(b + k*K
                  + j));
          }
      }
    }
    return C;
}
```

Next, we aim to output a Compute_time.txt file in the main.cpp file using a for loop, documenting the impact of matrix sizes on computation speed.

```
/* main.cpp */
#include <fstream>
std::ofstream outfile2("Compute_time.txt");
int size[5] = {10, 100, 1000, 2000, 4000};
for(int i = 0; i < 5; i++){   //here something is changed
    const size_t n = size[i];

      if (outfile2.is_open()) {
                outfile2 << n << "x" << n << " prod computed in
                    time_mult " << time_mult.count()<< "s for NKM-
                    Loop" << std::endl;
                  outfile2 << " Gflop/s "<< 2*n*n*n/time_mult.
                       count()/1.e+9 << std::endl;
          } else {
                  std::cerr << "Unable to open file for writing.
                       " << std::endl;
          }

}
 outfile2.close();
```

```
2000x2000 prod computed in time_mult 5.71944s for NKM-Loop
 Gflop/s 2.79748
2000x2000 prod computed in time_mult 5.74069s for KNM-Loop
 Gflop/s 2.78712
2000x2000 prod computed in time_mult 0.938238s for NKM-Loop
   with OpenMP
 Gflop/s 17.0532
2000x2000 prod computed in time_mult 0.116797s for BLAS
 Gflop/s 136.99
4000x4000 prod computed in time_mult 34.4119s for NKM-Loop
 Gflop/s 3.71965
4000x4000 prod computed in time_mult 47.9739s for KNM-Loop
 Gflop/s 2.66812
4000x4000 prod computed in time_mult 7.18593s for NKM-Loop
   with OpenMP
 Gflop/s 17.8126
4000x4000 prod computed in time_mult 0.700567s for BLAS
 Gflop/s 182.709
```

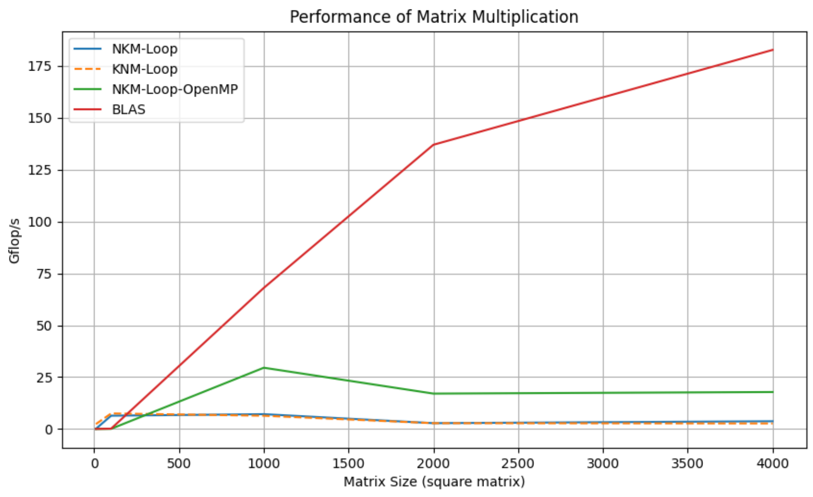
We also need to use a Python script (plot.py) to implement the plotting.

```
# plot.py #
import re
def read_data_from_txt(file_path):
    methods = ["NKM-Loop", "KNM-Loop", "NKM-Loop-OpenMP", "
       BLAS"]
    data = {method: {"sizes": [], "gflops": []} for method in
       methods}

    with open(file_path, 'r') as file:
        lines = file.readlines()

    for i in range(0, len(lines), 2):
        match = re.search(r'(\d+x\d+)', lines[i])
        size = int(match.group(1).split('x')[0])

        time_match = re.search(r'time_mult (\S+)s for (\S+)',
           lines[i])
        method = time_match.group(2)

        gflops = float(lines[i+1].split()[1])

        data[method]["sizes"].append(size)
        data[method]["gflops"].append(gflops)

    return list(set([item for sublist in [data[method]["sizes"
        ] for method in methods] for item in sublist])), data
```

3 Results and analysis

3.1 Arithmetics speed

In the following graph, we plot the computation speeds of four matrix multiplication methods as a function of matrix size. The x-axis represents different matrix sizes, while the y-axis represents computation speed (in floating point operations). The dgemm function provided by the OpenBLAS library is the fastest. The parallel computation offered by OpenMP shows a significant speedup, but as the matrix size exceeds 1000x1000, this improvement becomes harder to achieve with increasing matrix size. In contrast, the dgemm function, when dealing with a matrix size of 4000x4000, is 49 times faster than the most optimal for loop "NKM".



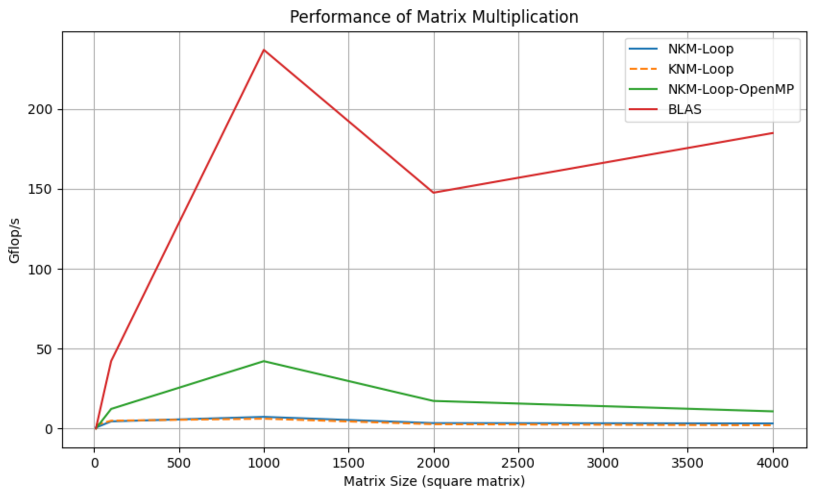
When the performance gain from using OpenMP plateaus or even decreases after a certain matrix size, it may indicate several potential scenarios :

- **Thread Saturation :** The number of threads spawned by OpenMP has reached the maximum capability of the hardware. For instance, if a CPU has 8 cores and OpenMP is already utilizing all of them, adding more threads won't provide any further speedup and can even lead to performance degradation due to overhead from context switching.
- **Memory Bandwidth Limitation :** As the size of the matrix increases, the amount of data that needs to be loaded and stored to/from memory also increases. There might be a point where the memory bandwidth becomes a bottleneck, limiting the performance benefits of parallelization.

- **Cache Contention :** For larger matrices, the data might not fit well in the CPU cache. Multiple threads could be competing for limited cache resources, leading to cache contention. This results in more cache misses and increased memory accesses, which can slow down computation.
- **False Sharing :** In multi-threaded applications, especially when using shared memory parallelism like OpenMP, false sharing can be an issue. If two threads update data that resides on the same cache line (even if they are updating different variables), it can cause performance degradation.

```
export OMP_NUM_THREADS=16
```

We allocated eight CPUs in the Ubuntu system within a virtual machine. We tried executing the program with 16 threads, and the results are shown in the following graph. We can observe that there is a gain for OpenMP, but it's marginal. Additionally, the efficiency of dgemm does not necessarily increase linearly with the size of the matrix. Moreover, we understand that, in general, hyper-threading does not always guarantee improved efficiency.



The performance of the dgemm function may be affected by the following factors :

- **Hardware Architecture :** Different CPU architectures might have varying impacts on the performance of matrix operations.
- **Memory Bandwidth :** Large matrix operations might be constrained by memory bandwidth limitations.
- **Cache Size and Organization :** The size and organization of the CPU cache can affect data locality, thereby influencing performance.
- **Parallelization and Vectorization :** Many modern BLAS implementations, like 'dgemm', try to fully exploit parallelization and vectorization for performance enhancement.

- **Matrix Size and Shape :** Matrices of certain sizes or shapes might be better suited for specific algorithms or hardware optimizations.
- **Other System Activities :** Other running processes or applications might compete with 'dgemm' for resources, affecting its performance.

Below are the results when viewing the CPU status using the "top" command while running a matrix of size 4000x4000. As we can see, the eight threads are almost fully utilized.

```
top - 21:14:07 up  6:22,  1 user,  load average: 3,08, 1,94, 1,31
Tasks: 249 total,   2 running, 247 sleeping,   0 stopped,   0 zombie
%Cpu(s): 97,8 us,  2,2 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :   7932,2 total,   3221,5 free,   2500,9 used,   2209,8 buff/cache
MiB Swap:   2048,0 total,   2048,0 free,      0,0 used.   5041,9 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   6513 zli2022   20   0 1641712 418592   3840 R 759,6   5,2   2:50.46 test_prod_openblas
```

Hyper-threading technology allows a single CPU core to execute two threads simultaneously. Ideally, this can enhance the CPU's throughput. However, in practical applications, the performance boost from hyper-threading might be affected by various factors. For instance, two threads might compete for the cache or other resources of the same core, leading to a drop in performance. Therefore, hyper-threading doesn't always result in a linear performance increase, and its efficiency depends on the specific workload and application context.

A Compilation details

```
make -j 8
```

The 'make -j 8' command is used for parallel compilation of projects. In this command, 'make' is a tool typically employed for automatically compiling and linking programs. The '-j' option informs 'make' to execute tasks in parallel. The number 8 specifies the number of tasks that can be run concurrently, allowing for 8 tasks to be executed at once. This command is especially beneficial on systems with multiple cores or threads, as it can significantly accelerate the compilation process by utilizing all available cores. However, setting the value for '-j' too high might overload system resources, especially memory, which could decrease efficiency or even cause the compilation to fail. It's generally recommended to set this value close to the number of CPU cores available.