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

HPC 4GMM 2021/2022

Members

PHAM Tuan Kiet

VO Van Nghia

Date

12 Dec, 2021

Contents

Contents **i**

1 Techniques **1**

1.1 Native dot 1

1.2 Spatial locality 1

List of Figures

1 Techniques

1.1 Native dot

We first mention here the original `native_dot` function. This function serves as an anchor (or base case) for performance comparision as well as for making sure we have the right result when using other techniques.

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++) C[i + ldc * j] += A[i + lda * k] * B[k + ldb * j];
```

Below is the output of `native_dot` for `M = 1`, `K = 2`, `N = 2`:

```
##
## Parallel execution with a maximum of 4 threads callable
##
## Scheduling static with chunk = 0
##
## ( 1.00  1.50 )
##
## ( 1.00  1.50 )
## ( 1.50  2.00 )
##
## Frobenius Norm   = 5.550901
## Total time naive = 0.000000
## Gflops           = inf
##
## ( 3.25  4.50 )
```

As

$$\begin{pmatrix} 1 & 1,5 \end{pmatrix} \begin{pmatrix} 1 & 1,5 \\ 1,5 & 2 \end{pmatrix} = \begin{pmatrix} 3,25 & 4,5 \end{pmatrix}$$

The result of this function is correct. We could move on to the next technique.

1.2 Spatial locality

Spatial locality refers to the following scenario: if a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In order to take advantages of this property, we notice that:

- In memory, `A`, `B`, `C` are stored in contiguous memory block.
- When using the index order `i`, `j`, `k`, we access `B` consecutively (as we access `B` by `B[k + ldb * j]`), but not `A` and `C`.
- Data from `A`, `B`, `C` are loaded in a memory block consisting of severals consecutive elements to cache. Thus, we could make use of spatial locality when reading data

continously.

From 3 points above, we decide to switch the index order to `k`, `j`, `i`. Now we see that both reading and writing operations on `C` are in cache, this brings us a critical gain in performance. In addition, reading operations on `A` are in cache too but those on `B` are not.

```
for (k = 0; k < K; k++)
  for (j = 0; j < N; j++)
    for (i = 0; i < M; i++) C[i + ldc * j] += A[i + lda * k] * B[k + ldb * j];
```

For comparision, we have a table below with small `M`, `K`, `N` (`OMP` indicates if we enable `Open MP` or not).

Technique	Time	Norm	Gflops	M	K	N	OMP
Naive	0	3.461352	Inf	4	8	4	FALSE
Saxpy	0	3.461352	Inf	4	8	4	FALSE

We have the frobenius norm of both techniques are 3,461352, which indicate we have the right computation result. In addition, calculating time is already significantly small (≈ 0 second in both methods) and the difference between these two can therefore be ommited.

However, if we set `M`, `K`, `N` to 2048. There will be a huge performance gain as in the table shown below.

Technique	Time	Norm	Gflops	M	K	N	OMP
Naive	82.764972	2.323362	0.207574	2048	2048	2048	FALSE
Saxpy	4.022001	2.323362	4.271473	2048	2048	2048	FALSE

Here, the `native_dot` function is approximately 21 times slower than the `saxpy_dot` function.