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

1 Techniques

1.1 Naive dot

We first mention here the original `naive_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 `naive_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 `OpenMP` or not).

Technique	Time	Norm	Gflops	M	K	N	OMP
naive	0	3.461352	Inf	4	8	4	F
saxpy	0	3.461352	Inf	4	8	4	F

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	62.267724	2.323362	0.275903	2048	2048	2048	F
saxpy	5.800958	2.323362	2.961557	2048	2048	2048	F

Here, the `naive_dot` function is approximately 11 times slower than the `saxpy_dot` function.

1.3 OpenMP parallelization

For parallelism, we add a directive `#pragma omp parallel for schedule(runtime)default(shared)` and `private(i, j, k)` depending on which variables we want to make private. A special clause `reduction(+ : norm)` is added to `norm` function as we want to sum all the `norm` from each thread to only one variable. A link to github with full source code will be provided at the end of the report. For performance comparision, we will show only one case here. More detailed study will be presented in the next sections.

Technique	Time	Norm	Gflops	M	K	N	Block	OMP	Threads	Schedule	Chunk
naive	62.267724	2.323362	0.275903	2048	2048	2048	4	F	4	static	0
saxpy	5.800958	2.323362	2.961557	2048	2048	2048	4	F	4	static	0
naive	33.076697	2.323362	0.519395	2048	2048	2048	4	T	4	static	0
saxpy	4.116676	2.323245	4.173238	2048	2048	2048	4	T	4	static	0

1.4 Cache blocking

The main idea of the cache blocking technique (or tiled) is breaking the whole matrices into smaller sub-matrices so the data needed for one multiplication operation could fit into the cache, therefore leads to a much faster calculation. Furthermore, if we enable `OpenMP`, the computation would be even faster as each sub-matrice is processed by a separate thread. However, if we set `BLOCK` size too small, the benefit of dividing matrix is overshadowed by the additional loops and operations. Meanwhile, a too large `BLOCK` size leads to an overfitting (data for one operation can not be fitted into the cache), and therefore a slower operation. The principal source code is shown below:

```
for (k = 0; k < K; k += BLOCK) {
  for (j = 0; j < N; j += BLOCK) {
    for (i = 0; i < M; i += BLOCK) {
      int kmin = fmin(K - k, BLOCK);
      for (kk = 0; kk < kmin; kk++) {
        int jmin = fmin(N - j, BLOCK);
        for (jj = 0; jj < jmin; jj++) {
          int imin = fmin(M - i, BLOCK);
          for (ii = 0; ii < imin; ii++) {
            C[(ii + i) + ldc * (jj + j)] +=
                A[(ii + i) + lda * (kk + k)] * B[(kk + k) + ldb * (jj + j)];
          }
        }
      }
    }
  }
}
```

In addition, we see in the table below that we have 1,71 times faster with the non `OpenMP` version and 1,79 times with `OpenMP`.

Technique	Time	Norm	Gflops	M	K	N	Block	OMP	Threads	Schedule	Chunk
naive	62.267724	2.323362	0.275903	2048	2048	2048	256	F	4	static	0
saxpy	5.800958	2.323362	2.961557	2048	2048	2048	256	F	4	static	0
tiled	3.387300	2.323362	5.071848	2048	2048	2048	256	F	4	static	0
naive	33.076697	2.323362	0.519395	2048	2048	2048	256	T	4	static	0
saxpy	4.116676	2.323245	4.173238	2048	2048	2048	256	T	4	static	0
tiled	2.298825	2.321977	7.473326	2048	2048	2048	256	T	4	static	0

1.5 BLAS

One last technique that is used in our code is calling the `cblas_dgemm` function which use the optimized BLAS implementation. This function is the fastest method even if other methods are *"cheated"* (by using `OpenMP`) as their implementation is optimized based on many factors: algorithms, software and hardware.

Technique	Time	Norm	Gflops	M	K	N	Block	OMP	Threads	Schedule	Chunk
naive	33.076697	2.323362	0.519395	2048	2048	2048	256	T	4	static	0
saxpy	4.116676	2.323245	4.173238	2048	2048	2048	256	T	4	static	0
tiled	2.298825	2.321977	7.473326	2048	2048	2048	256	T	4	static	0
blas	0.375638	2.323362	45.735173	2048	2048	2048	256	T	4	static	0

2 Sequential

In this section, we fix `OMP_NUM_TREADS=1` for each run and vary matrix size and blocking size (We don't care about schedule and chunk size because there is only 1 thread).

2.1 M = K = N

First, we consider the case where M and K and N are all equal and equal to a 2^s. The blocking size is also supposed to be in a form of 2^t. In addition, as `OMP_NUM_TREADS=1` is essentially the same logic as non `OpenMP` version, we will include a non `OpenMP` result for studying how the overhead time of `OpenMP` impact the overall performance.

```
## Warning in py_to_r.pandas.core.frame.DataFrame(x): index contains duplicated
## values: row names not set
```

Technique	Time	Norm	Gflops	M	K	N	Block	OMP	Threads	Schedule	Chunk
naive	0.000002	4.746709	0.008389	2	2	2	1	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	1	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	1	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	1	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	2	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	2	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	2	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	2	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	4	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	4	T	1	static	0
tiled	0.000001	4.746709	0.013422	2	2	2	4	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	4	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	8	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	8	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	8	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	8	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	16	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	16	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	16	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	16	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	32	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	32	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	32	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	32	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	64	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	64	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	64	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	64	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	128	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	128	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	128	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	128	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	256	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	256	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	256	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	256	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	512	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	512	T	1	static	0
tiled	0.000002	4.746709	0.007457	2	2	2	512	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	512	T	1	static	0
naive	0.000002	4.746709	0.008389	2	2	2	1024	T	1	static	0
saxpy	0.000001	4.746709	0.016777	2	2	2	1024	T	1	static	0
tiled	0.000001	4.746709	0.016777	2	2	2	1024	T	1	static	0
blas	0.000015	4.746709	0.001065	2	2	2	1024	T	1	static	0
naive	0.000003	3.430390	0.044739	4	4	4	1	T	1	static	0
saxpy	0.000001	3.430390	0.134218	4	4	4	1	T	1	static	0
tiled	0.000001	3.430390	0.107374	4	4	4	1	T	1	static	0
blas	0.000016	3.430390	0.008013	4	4	4	1	T	1	static	0
naive	0.000003	3.430390	0.044739	4	4	4	2	T	1	static	0
saxpy	0.000001	3.430390	0.134218	4	4	4	2	T	1	static	0
tiled	0.000002	3.430390	0.067109	4	4	4	2	T	1	static	0
blas	0.000016	3.430390	0.008013	4	4	4	2	T	1	static	0
naive	0.000003	3.430390	0.044739	4	4	4	4	T	1	static	0
saxpy	0.000001	3.430390	0.134218	4	4	4	4	T	1	static	0
tiled	0.000001	3.430390	0.134218	4	4	4	4	T	1	static	0
blas	0.000016	3.430390	0.008013	4	4	4	4	T	1	static	0
naive	0.000003	3.430390	0.044739	4	4	4	8	T	1	static	0