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 = 0.000001
## Gflops           = 0.008389
##
## ( 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 (s)	$\|.\|_F$	$GFlop/s$	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 (s)	$\|.\|_F$	$GFlop/s$	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 (s)	$\|.\|_F$	$GFlop/s$	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); // in case K is not divisible by BLOCK
      for (kk = 0; kk < kmin; kk++) {
        int jmin = fmin(N - j, BLOCK); // in case N is not divisible by BLOCK
        for (jj = 0; jj < jmin; jj++) {
          int imin = fmin(M - i, BLOCK); // in case M is not divisible by 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 (s)	$\|.\|_F$	$GFlop/s$	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 (s)	$\Vert.\Vert_F$	$GFlop/s$	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 the matrix size. (We don't care about schedule because there is only 1 thread).

For the sake of simplicity, 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.

In the graph below, we see that the fastest method is no doubt `blas` method, followed by `tiled` (with a block of 256). The third fastest method is `saxpy` and the slowest is `naive`. This is aligned with what we see in the section 1. In addition, the time for calculating matrices whose size is less than $2^{10} = 1024$ is around 5 s for all methods. This could be explained by the fact that these matrices could be fitted entierly into the cache, which leads to a significant drop in computation time.

Another property that could be interesting is the version with `OpenMP` is close or even faster than the non `OpenMP` version regardless the overhead of parallelization. This could be explained by many factors [1] [2], but the most significant one is As `OpenMP` is just API specification and C compilers are free to implement it in any way they want as long as they respect the specification, many compilers (notably modern `gcc` and `clang`) are smart enough to treat `OpenMP` version of only 1 thread the same as the sequential version. Therefore, we only see a small difference between each run. If we run both versions enough times, the average time of both will be the same.

[1] https://stackoverflow.com/questions/22927973/openmp-with-single-thread-versus-without-openmp

[2] https://stackoverflow.com/questions/2915390/openmp-num-threads1-executes-faster-than-no-openmp

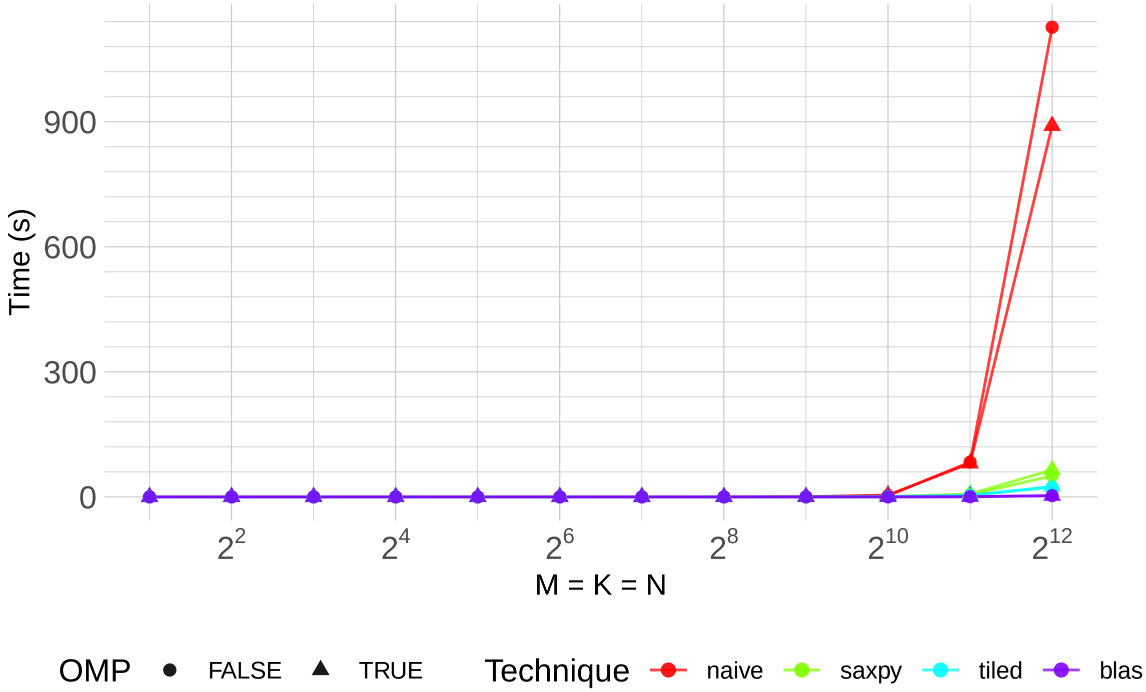


Figure 1: Computation time in function of matrix size and technique

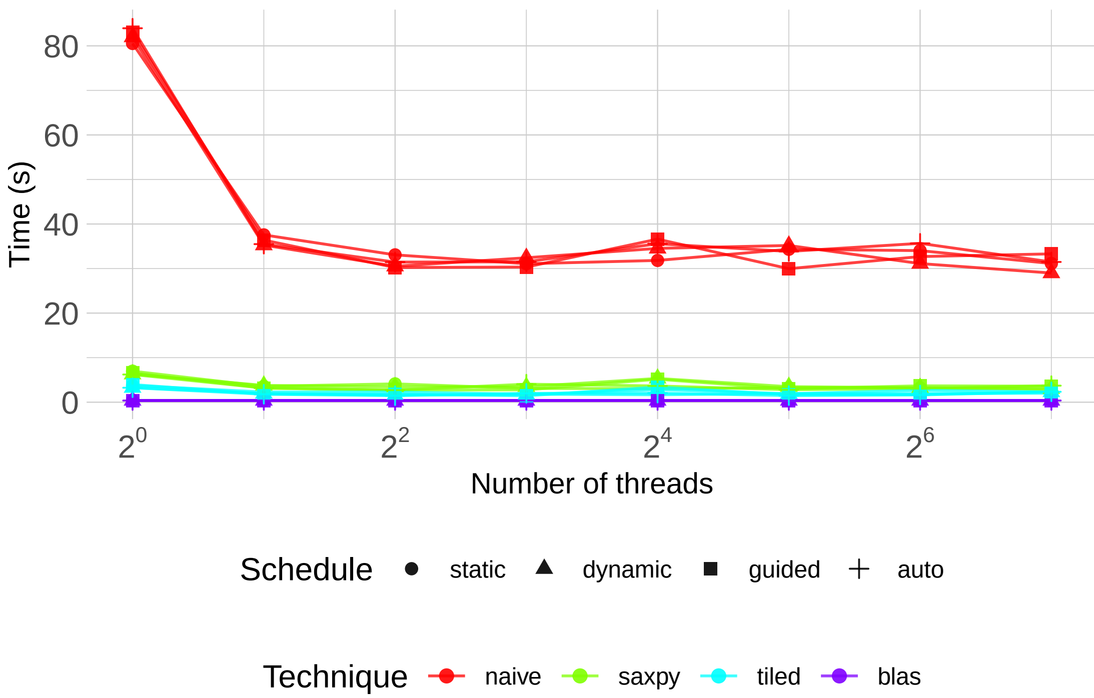
3 Threading

Right now, we want to see the true power of parallelism, we will fix matrix size to `M` = `K` = `N` = 2048, `block` to 256, and vary the number of threads.

We see that although the distance between are relatively small, `blas` method is still the fastest regardless the number of threads. It shows that in order to achieve high speed computation, we have to not only parallelize, but also make improvements on multiplication algorithms, memory accesses and even use assembly instructions.

In addition, the 4 schedule lines of each technique are overlapping each others and there are only very small difference in term of computational time. Except `blas` method which is not affected by the schedule options, the phenomenon happened because our problem (matrix multiplication) has a nearly the same workload at each iterations. That means the first iteration will take almost the same as the last iteration or any other iterations. For each schedule:

- `static` evenly-divides the total workloads into each threads, which is the best schedule for our problem.
- `dynamic` and `guided` are designed for different situation, where each iteration takes different amount of time to finish their work. There is overhead compared to `static`, however, it does not have big effect on overall performance as our matrices are not too big.
- `auto` lets the compiler choose how to schedule and divide work among threads, so it is compiler-specific. For example, `gcc` maps `auto` to `static` [3], at a consequence, we see a similar pattern with `static`.

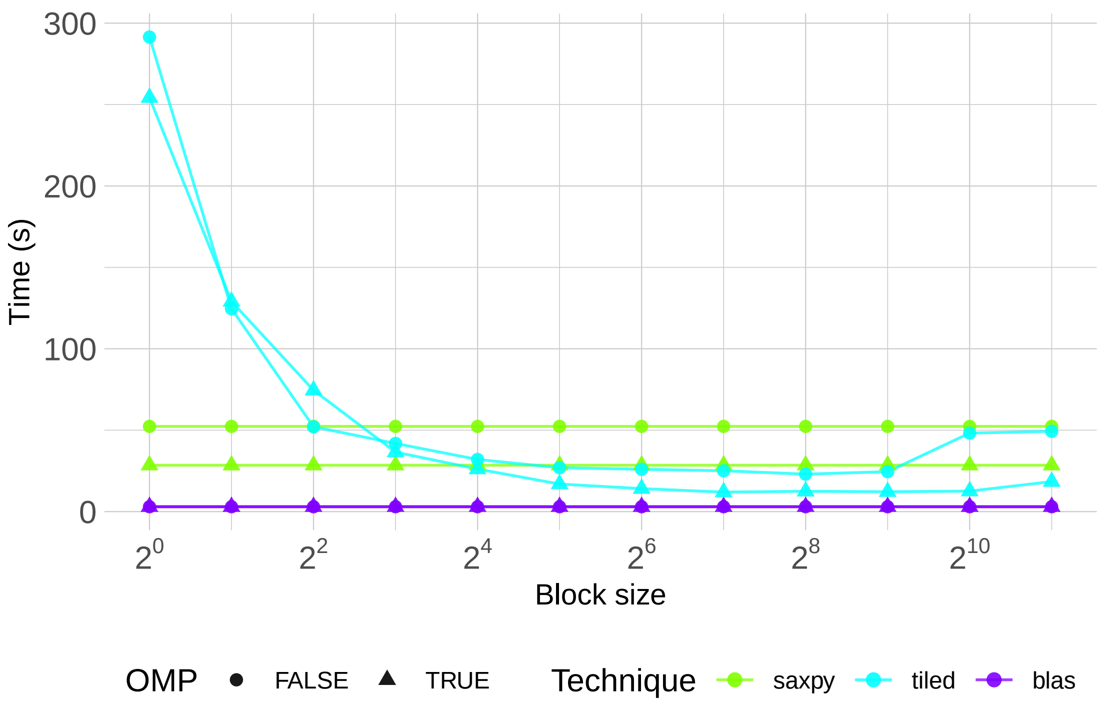


Finally, more threads **doesn't** always mean better performance. After we increased thread to 2, time taking for one multiplication fluctuates but does not have any real decline. The reason is there are only 2 physical cores on this computer, when the number of threads goes up too high, the overhead in creating and synchronize threads will overshadowed any benefits we gain.

[3] https://github.com/gcc-mirror/gcc/blob/61e53698a08dc1d9a54d785218af687a6751c1b3/libgomp/loop.c#L195-L198

4 Blocking

In the last section, we will concentrate ourselves on the impact of `BLOCK` size to overall performance. We will fix `M` = `K` = `N` = 2048, number of threads to 4, `static` schedule and vary the `BLOCK` size.



We see clearly that as `BLOCK` size grows, the performance becomes better but get worse after `BLOCK` size grows to approximately 2^10. As explained in the section 1, `BLOCK` should not be too small and neither too large.

5 Conslusion

Throughout the report, it is shown clearly that there are many techniques to speed up our code. Depending on each problem and resources, we could also have a combination of all these techniques as well as having a detail benchmark to achive the best possible performance.

Appendix A: `blas3.c`

We also like to include a part which details our experiences when working on this report.

Mistake between `M` and `N`

Initially, when we set `M` $= 1$, `K` $= 2$, `N` $= 2$, we have this output:

```
##
## 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   = nan
## Total time naive = 0.000000
## Gflops           = inf
##
## ( 3.25  0.00 )
##
## Frobenius Norm   = nan
## Total time saxpy = 0.000000
## Gflops           = inf
##
## ( 3.25  0.00 )
##
## Frobenius Norm   = nan
## Total time tiled = 0.000001
## Gflops           = 0.008389
##
## ( 3.25  0.00 )
##
## Frobenius Norm   = 3.250000
## Total time BLAS  = 0.000018
## Gflops           = 0.000447
##
## ( 3.25  0.00 )
```

The result should be $\begin{pmatrix} 3,25 & 4,5 \end{pmatrix}$. Furthermore, the function signatures of `norm` and `printarray` are:

```
double norm(int nrow, int ncol, int ld, double *A);
void print_array(int nrow, int ncol, int ld, double *A);
```

However, we found below this piece of code that suggests there maybe a mistake between `M` and `N`

```
printf("Frobenius Norm   = %f\n", norm(N, M, ldc, c));
```

```
// ...
print_array(M, N, ldc, c);
```

When we fixed `M` and `N` and rerun the code again. It show this output

```
##
## 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 )
##
## Frobenius Norm   = 7.155636
## Total time BLAS  = 0.000019
## Gflops           = 0.000425
##
## ( 3.25  6.38 )
```

We succesully fixed the "naive" method but there are still something "weird" with `cblas_dgemm` output. We dig deeper into the document and found that there is something wrong with

```
int ldа = N + 1;
int ldb = K + 1;
int ldc = N + 1;

double *a = (double *)malloc(lda * K * sizeof(double));
double *b = (double *)malloc(ldb * M * sizeof(double));
double *c = (double *)malloc(ldc * M * sizeof(double));

// ...

cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha, a, lda,
            b, ldb, beta, c, ldc);
```

Here, `a` is supposed to be a `double` pointer whose size is `M * K` but the code above allocated a pointer with the size of `(N + 1) * K`. According to the document, when using with `CblasColMajor` and `CblasNoTrans`, `lda` should be the number of elements in one column, so `lda` is equal to `M`. After editing the code to

```
int lda = M;
int ldb = K;
```

```
int ldc = M;

double *a = (double *)malloc(lda * K * sizeof(double));
double *b = (double *)malloc(ldb * N * sizeof(double));
double *c = (double *)malloc(ldc * N * sizeof(double));
```

We got the correct result as shown in the section 1.

Minor issues

When trying to change the schedule, the schedule value is switched. According to the `OpenMP` document, `guided` should be 3 and `auto` is 4 instead of vice versa.

Finally, we add a

```
free(a);
free(b);
free(c);
```

in the end for preventing memory leaks.

Source code

Full source code could be found on github.