

# CS 5220 Homework 1 Report

Name: Xingze Li

NetId: xl834

Date: 24/Feb/2023

---

## Introduction

In this homework, the goal is to write single core matrix multiplication code for square matrices as fast as possible. The vanilla mathematical principle is quite simple in matrix formula:

$$C[i,j] = A[i,:] * B[:,j]$$

The simplest way to get matrix  $C$  is to use three loops to iterate A and B:

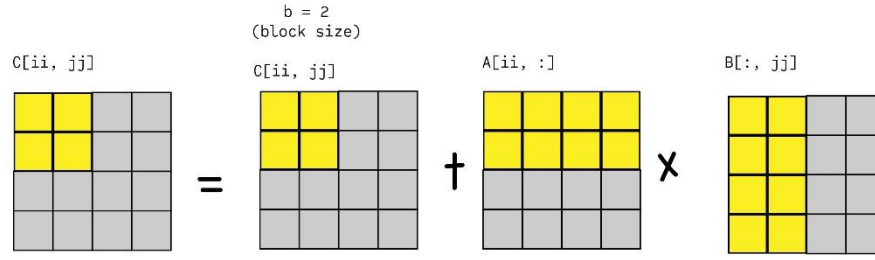
```
void simpleGEMM(double *A, double *B, double* C, int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            C[i + j * n] = 0.0;
            for(int k = 0; k < n; k++) {
                C[i + j * n] += A[i + k * n] * B[k + j * n];
            }
        }
        ...
    }
}
```

However, this code implementation has far more room to be augmented. There are several ways to speed up the matrix multiplication: blocking, repack, realign, change the loop order and micro-kernel. I try these methods and their combinations to enhance the performance. The input matrix A is column major and B is row major and output C is column major, they are all one-dimension matrices, which is better than general two-dim matrix implementation(illustrated in detail in implementation section).

## Implementation

### 1. Blocking

In this method, I split the entire matrix dealing into even sub-matrix multiplication and addition. For example, I have input matrices with size  $N * N$ , I define a sub-matrix length  $b$ :



ii, jj, kk denote block indices while i, j, k denote element indices

```
// C = C + A * B
// b = n / N (where b is the block size)
for ii = 1 to N:
  for jj = 1 to N:
    for kk = 1 to N:
      C[ii, jj] += A[ii, kk] * B[kk, jj]
```

In the figure above, splitting the  $4 \times 4$  matrix into 4 equal sized matrices, implement multiplication on these small blocks in the same way as elementwise multiplication. It should be noticed when the input matrix size is not the multiple of block size we choose, we should regulate the parameter carefully to deal the corner cases.

In terms of computing the optimal block size, I consider the cache size L1 and L2 at first. I use L1 as bound and compute the block size with:

$$3 * N^2 \leq L1 \quad (N \text{ is block size, } L1 \text{ is cache capacity})$$

which has a more general formula:

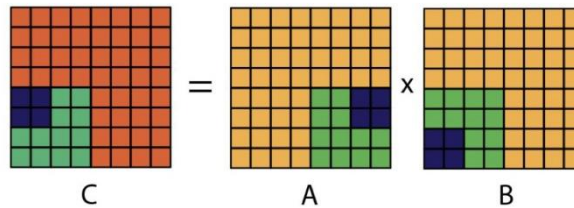
$$mn + mk + kn \leq C \quad (A \in R^{m \times k}, B \in R^{k \times n}, C \in R^{m \times n})$$

since we want the entire cache contains all 3 sub blocks at the same time, which reduce the time of accessing memory. We have  $L1 = 32KiB$ , therefore  $N \leq \sqrt{\frac{L1}{3}} = 37B$ .

For  $L2 = 512KiB$ ,  $N \leq \sqrt{\frac{L2}{3}} = 148B = 4 * 37B$ . However, influenced by compiler optimization, instruction set and memory access, the theoretical doesn't acquire the best performance. After having tried block size of 37, 32, 16, 4 and values around these. The optimal block size is 33. The evaluation is in performance section. Another matter is dealing the bound condition, I make the minimum of matrix size  $lda$  and left size  $lda - i, lda - j, lda - k$  as three bounds of blocking part.

## 2. Multi-level Blocking

Similarly, on the basis of blocking, there is multi-level blocking, splitting the sub block one more time to implement matrix multiplication:



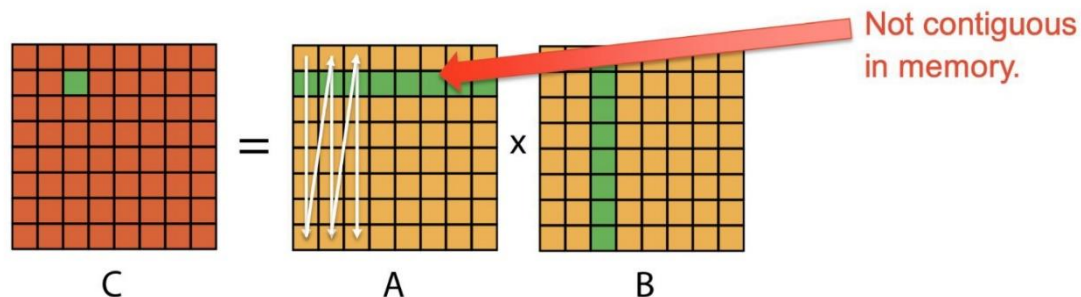
Since this method shares the same principle with blocking, the implementation steps

are nearly identical. There is a point need to pay attention to, when the top-level split encounter the situation that the block size is larger than the length left, it shouldn't be split into sub blocks but should deal it with vanilla blocking matrix multiplication.

The theoretical block size of top-level split is 148. However, the optimal is far less than it, which is illustrated in performance section.

### 3. Repack

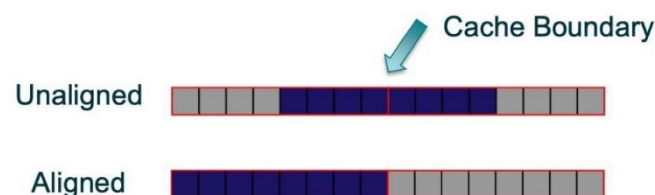
In naïve matrix multiplication, since we iterate the inner loop through A in column major order which is incontiguous in memory since each row of A is store in memory (which is indicated by every time we access element of A using  $A + i + j * lda$ ). So in order to access memory of A in a contiguous way and take advantage of hardware prefetching. I repack the matrix of A (only A since we access B in row-order which matches the order it saved in memory) by transposing it. Figure below shows the motivation of repacking.



The transposing is an additional step compared with naïve GEMM, however, its benefits overweight the extra time cost since transposing a matrix in general way cost  $O(n^2)$  while matrix multiplication is  $O(n^3)$ .

### 4. Realign

The data in memory not always aligned, as figure shown in below:

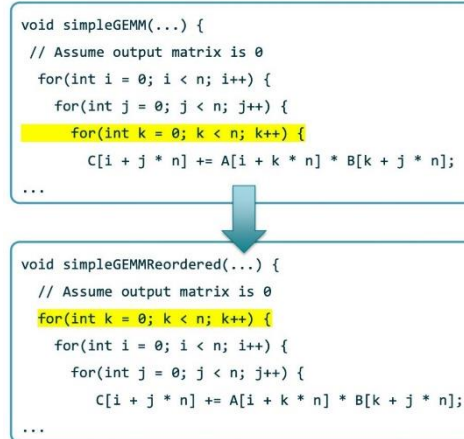


Therefore, in order to access it contiguously when we access a bunch of contiguous elements (when ensuring the start of the segment is aligned with a cache boundary) later, I make the memory block aligned, which also benefits SIMD when it allocates or access contiguous segments of 512bit/8 double words in memory. By using code piece `_mm_malloc(<bytes>, 64)` we can align 64-byte memory.

### 5. Loop Order Optimization

Considering the naïve matrix multiplication, it has to access element of A iteratively in column order, which is not contiguous in memory, however, if we change the order

of these loops, we can avoid this situation. There are 6 orders in total. We consider how many times it needs to jump discontinuously in memory. It should be noted that it is different for one-dim matrix access and two-dim matrix access, i.e., number of jumps needed are calculated in different way. In general, i, k, j order can be the optimal way since its  $n^2$  jumps needed.



```
void simpleGEMM(...) {
    // Assume output matrix is 0
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            for(int k = 0; k < n; k++) {
                C[i + j * n] += A[i + k * n] * B[k + j * n];
            }
        }
    }
    ...
}

void simpleGEMMReordered(...) {
    // Assume output matrix is 0
    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                C[i + j * n] += A[i + k * n] * B[k + j * n];
            }
        }
    }
    ...
}
```

Once we implement loop order optimization, we should change the multi-level order when we implement it, its inner multiplication should change. Moreover, once we change the order, we can repack B in column major to match the new order.

## 6. Micro-kernel

In this method, I use AVX intrinsics to vectorize data, by declaring input as a 512 bits vector, which containing 4 double words fitting a 2\*2 block exactly. Then load these data from memory and finish computation by `_mm512_madd_pd` or `_mm512_add_pd` plus `_mm512_mul_pd`. Then put this micro-kernel into bottom level block in multi-level blocking.

## Performance

I have tried many single and combinations of these methods. Acquiring many different performance data, however, it is frustrating for me to get merely slight enhance on performance though I struggled to push the performance.

For blocking, the result is 5.5% of peak performance in average when the block size is 33, larger than other block size like theoretically optimal 37 or 16 etc.

For multi-level blocking, the result is 4.66% when set the top-level block size to 32 and sub block size to 4, which I think is problematic in terms of my implementation. Although spending lots of time try to fix it but failed. Then I tried different size combination like 64-4, 128-4, 128-16 etc. The sub block size of 4 is always better than other size option. Only get results around 4%-5%.

For repack the matrix and change the order in i, k, j order combined with blocking, I get the result of 8.91%, which is relatively larger than only blocking.

For realign the matrix with multi-level blocking, I get the results 9.05%, 8.89% and 8.84% corresponding to block size sets of 32-4, 64-4, 128-4.

For AVX combined with multi-level blocking, I get result of 9.02% with block size

32-4 and 9.04% with block size 64-4, which is also unreasonable. Materials indicate that using AVX should enhance the performance significantly. Or I may implement it in a wrong way.

## Reference

<https://www.iarpa.gov/images/research->

[programs/AGILE/Baseline\\_Computer\\_System\\_Information.pdf](#)

[19.1 C++ Compiler Developer Guide and Reference \(intel.com\)](#)

[PHiPAC Fast Matrix Multiply Home Page \(berkeley.edu\)](#)

[performance - 4x4 double precision matrix multiply using AVX intrinsics \(inc. benchmarks\) -](#)

[Code Review Stack Exchange](#)