CSE260 Assignment1 - Matrix Multiply Routine

Author: Le Yang, Zehui Jiao

Q1. Result

Q1.a. Performance of Blislab

N	Peak GF
32	15.92
64	21.16
128	23.67
256	25.715
511	25.64
512	25.71
513	24.47
1023	25.63
1024	25.965
1025	23.99
2047	26.005
2048	25.945

Besides the result in the table, for the larger matrix sizes (Average for N > 512), we got our geometric mean at 25.15.

Q1.b. Performance Figure

The figure above shows performance of different methods. The x-axis is from 32 to 2048. For naive method, we only include N <= 1025 because it's too slow. And we repeat the experiment 20 times for every size and compute the average.

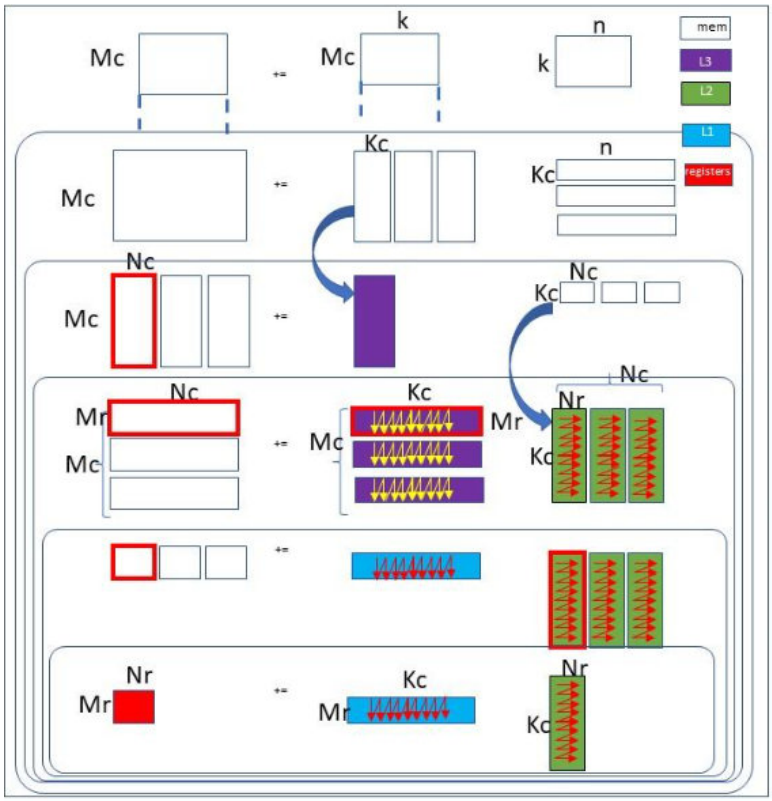
Q2. Anaylsis

Q2.a. How Does the Program Work

Skeleton Code

- benchmark.c - main driver program
- my_dgemm.c - this has the skeleton code for blislab.
  - bl_dgemm(loop5-loop3)
  - bl_macro_kernel(loop2-loop1)
  - packA_mcxkc_d and packB_kcxnc_d(packing routine)
- bl_dgemm_ukr.c - this has the skeleton code for blislab(micro kernel).
  - bl_dgemm_avx_256_4_4
  - bl_dgemm_avx_256_4_8

Workflow




```
Loop5: for ic = 0; ic < m; ic += Mc
         partition C into panels Cj of Mc x n
         partition A into blocks of Mc x n
Loop4:   for pc = 0; pc < k; pc += Kc
           part. A (Mc x n) -> Mc x Kc (Ap)
           pack Ap into subpanels Mr x Kc
           part. B into panels of Kc x n (Bp)
Loop3:     for jc = 0; jc < n; jc += Nc
             pack Bp into subpanels Kc x Nr
             part. Cj into panels of Mr x Nc
             // macrokernel
Loop2:       for ir = 0; ir < Mc; ic += Mr



Loop1:           for jr =0; jr < Nc; jr += Nr



                   // microkernel
Loop0:             for kr=0; kr < Kc; kr++
```

Based on J. Huang, R van de Geijn, "BLISLab: A Sandbox for Optimizing GEMM", FLAM Working Note #80, 8/31/2016 1

The figure above describes the hierarchy of how this program works. We need to divide a whole matrix into many small matrixes. In the Loop5, we use a parameter Mc to partition the matrix A and C. So we have $\frac{m}{Mc}$ iterations and every panel of $Mc \times n$. In the loop4, we use Kc which is the number of rows in block A and number of columns in block B to process in every iteration. It's about $\frac{k}{Kc}$ iterations. And In this loop, we want to fit the Ap into L3 cache. So $Mc * Kc$ should be less or equal to the size of L3 cache(30 MB). In the loop3 we will use Nc to divide the block C and block B. And In this loop, we want to fit the Bp into L2 cache. So $Kc * Nc$ should be less or equal to the size of L2 cache(256 KB). In the next step, we will call a macro kernel to pack panel A and panel B. In the packing routine, we use unrolling to accelerate the process. After packing, we will call a microkernel to calculate the multiplication of the subpanels. And we want to fit subpanel of A in L1 cache. So $Mr * Kc$ should be less or equal to the size of L1 cache(32 KB). And we implement the avx micro kernel, which could hold multiple doubles at the same time. So we could accelerate the computation process.

Q2.b. Development Process

Basic Optimization

Using Pointers

In this project, our compiler optimization is level 3, which is as same as the blislab. So actually we don't really need to use pointers. But we still use it. Using pointer could reduce the overhead of computation of the address where a matrix exists. We only need to increase the pointer instead of computation of add and multiply.

Loop Unrolling

As known, loop could increase the overhead. So we want to use loop unrolling to expand the inner loop. We choose 4 as the unroll factor.

```
for (i = 0; i < DGEMM_MR; i += 4) {
        *packA = *a_ptr[i]++;
        *(packA + 1) = *a_ptr[i + 1]++;
        *(packA + 2) = *a_ptr[i + 2]++;
        *(packA + 3) = *a_ptr[i + 3]++;
        packA += 4;
      }
```

But we also need to consider the edge case. If the number of row or column is not multiple of 4, then the last subpanel is less than 4, and we will handle this special situation.

Cache Optimization

Packing

We believe packing has two advantages. First, it could fit the proper matrix into the corresponding cache. So we need to think about the packing size. The other advantage is that we could redesign the memory order of the matrix. If the memory is aligned, then we could have better cache utilizatoin.

Padding

To avoid dealing with the different sizes of blocks in the micro kernel, we choose to pad the matrix to the multiple of `DGEMM_MR` and `DGEMM_NR`. So we don't need to modify the avx function. In order to keep the correct result, we should pad with zero.

Microkernel Optimization

Register Keyword

When implementing SIMD using AVX2, we set `register` keyword in front of each `__m256d`. The `register` keyword can hint to compiler that a given variable can be put in a register.

FMA Instruction

AVX introduces fused multiple add operation (FMA), which speed up computation. Here we use `_mm256_fmadd_pd` to apply FMA to increase speed.

SIMD based on AVX2

We use the broadcast method to implement SIMD based on AVX2.
On t2.micro, there are 16 AVX registers and each of them is 256-bits which can store 4 double precision numbers(VLEN=4). When determining to appropriate value for Mr and Nr, we set Kc=512, Mc=1024, Nc=32.

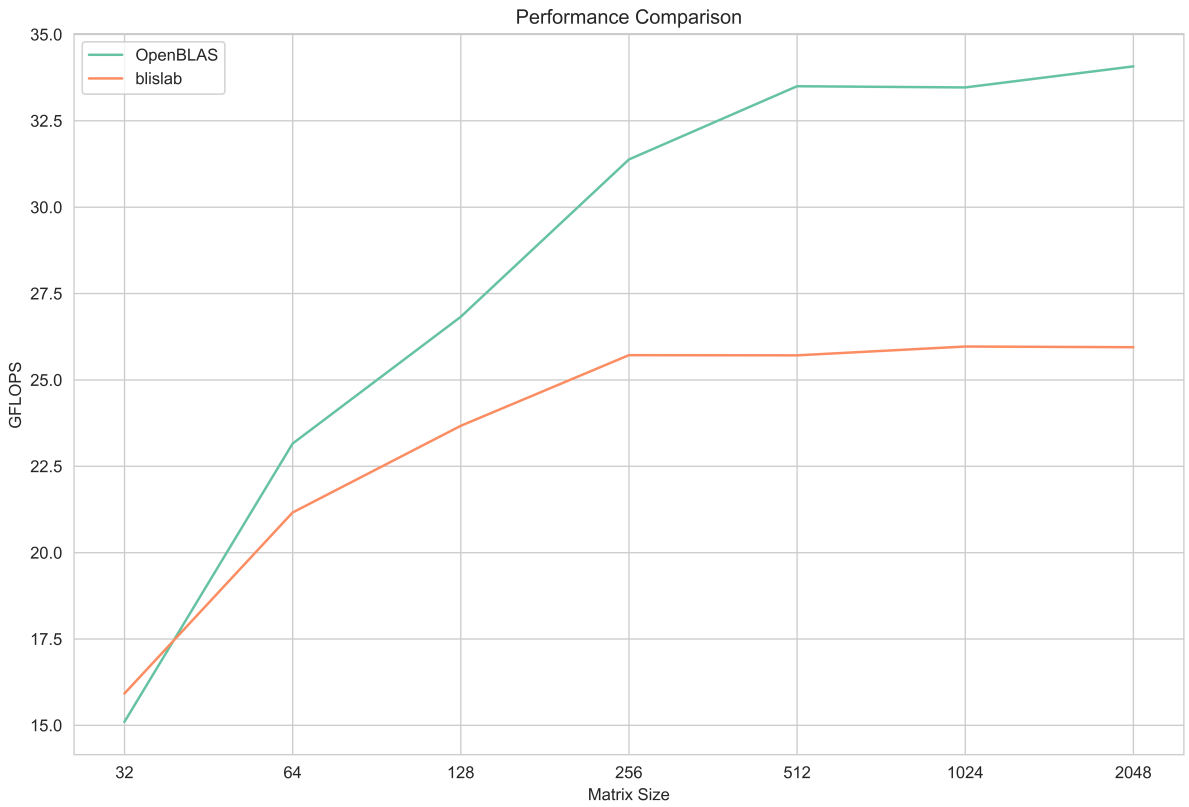
In the first attempt, we set Mr=4 and Nr=4. In each loop, we broadcast 4 elements in the subpanel of A to 4 vectors, each of which has length at 4; also we load 4 elements in subpanel of B to 1 vectors with length at 4. After the experiment, We get the geo mean result at 14.55 when the size of matrix range from 32 to 1025.

In the second attempt, we set Mr=4 and Nr=8. In each loop, we broadcast 4 elements in the subpanel of A to 4 vectors, each of which has length at 4; also we load 8 elements in the subpanel of B to 2 vectors with length at 4. After the experiment, We get the geo mean result at 24.44 when the size of matrix range from 32 to 1025.

From the results and the code, we find that in the first attempt we only use 9 registers in the microkenel: 4 registers for the subpanel of C, and 4 registers for the subpanel of A and 1 register for the subpanel of B; while in the final attempt we use 14 registers in the microkernel, 8 registers for the subpanel of C, and 4 registers for the subpanel of A and 2 registers for the subpanel of B. Therefore, we set MR=4 and NR=8.

Q2.c. High Level Irregularities

Multiple of 32



Finally, we choose `DGEMM_MR` as 4 and `DGEMM_NR` as 8. So we could pay attention to the multiple of 32. Our matrix could be divided into proper small panels. In this situation, GFLOPS of our method is increasing with matrix size and has an upper limit. When matrix size is larger than 256, the performance of our method is about 25-26 GFLOPS. We could say the best performance is about 26 GFLOPS.

As for OpenBLAS, the performance also increases with the size.

Multiple of 32 ± 1

N	Peak GF
511	25.64
512	25.71
513	24.47
1023	25.63
1024	25.965
1025	23.99
2047	26.005
2048	25.945

As for OpenBLAS, we have the similar situation. The performance is not ideal when the size is multiple of 32 ± 1.

Q2.d. Supporting Data

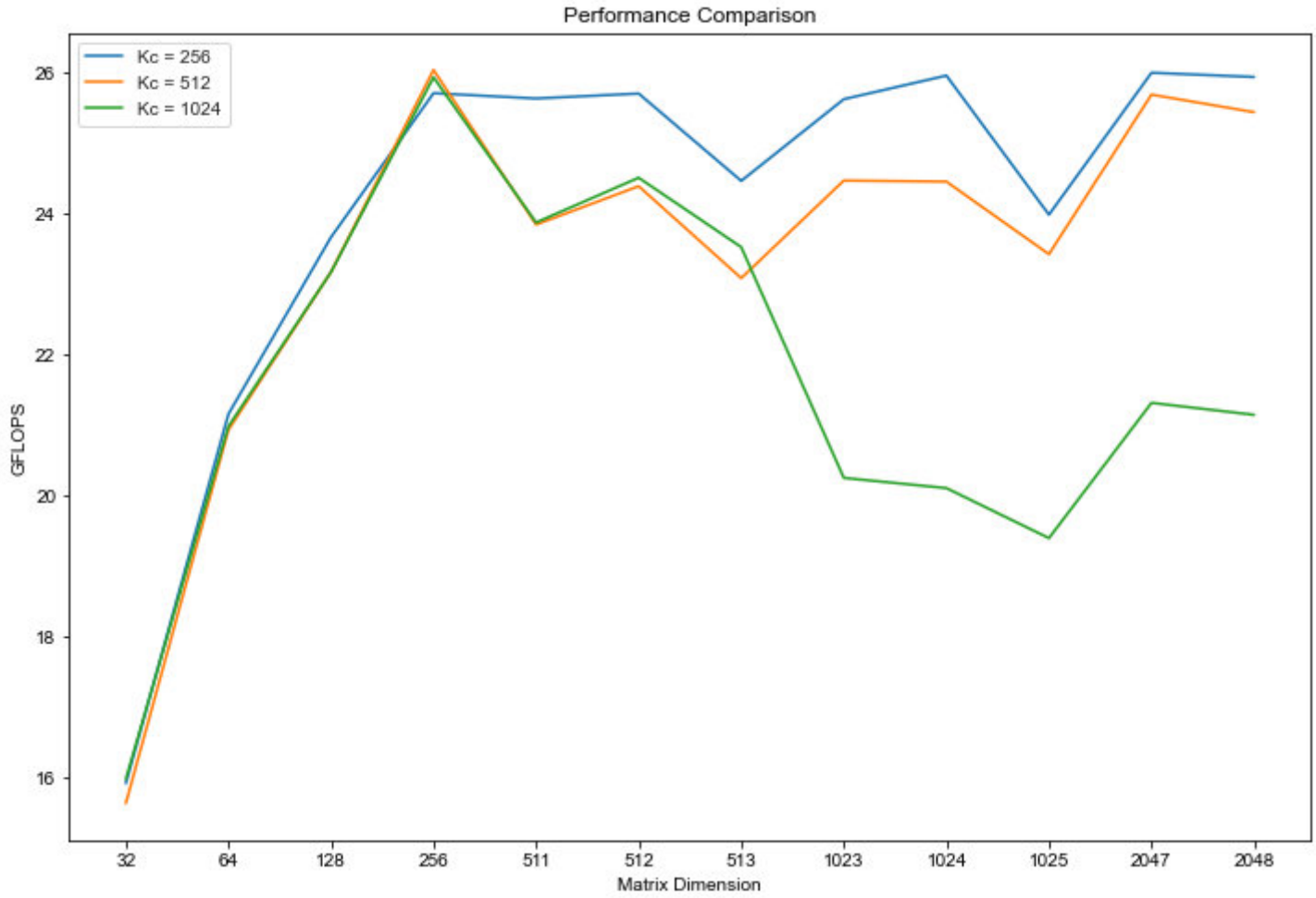
Using the command `getconf -a | grep CACHE`, we can find that the size for the three level caches of our system:

- L1 Cache: 32KB
- L2 Cache: 256KB
- L3 Cache: 30MB

Parameter Choosing

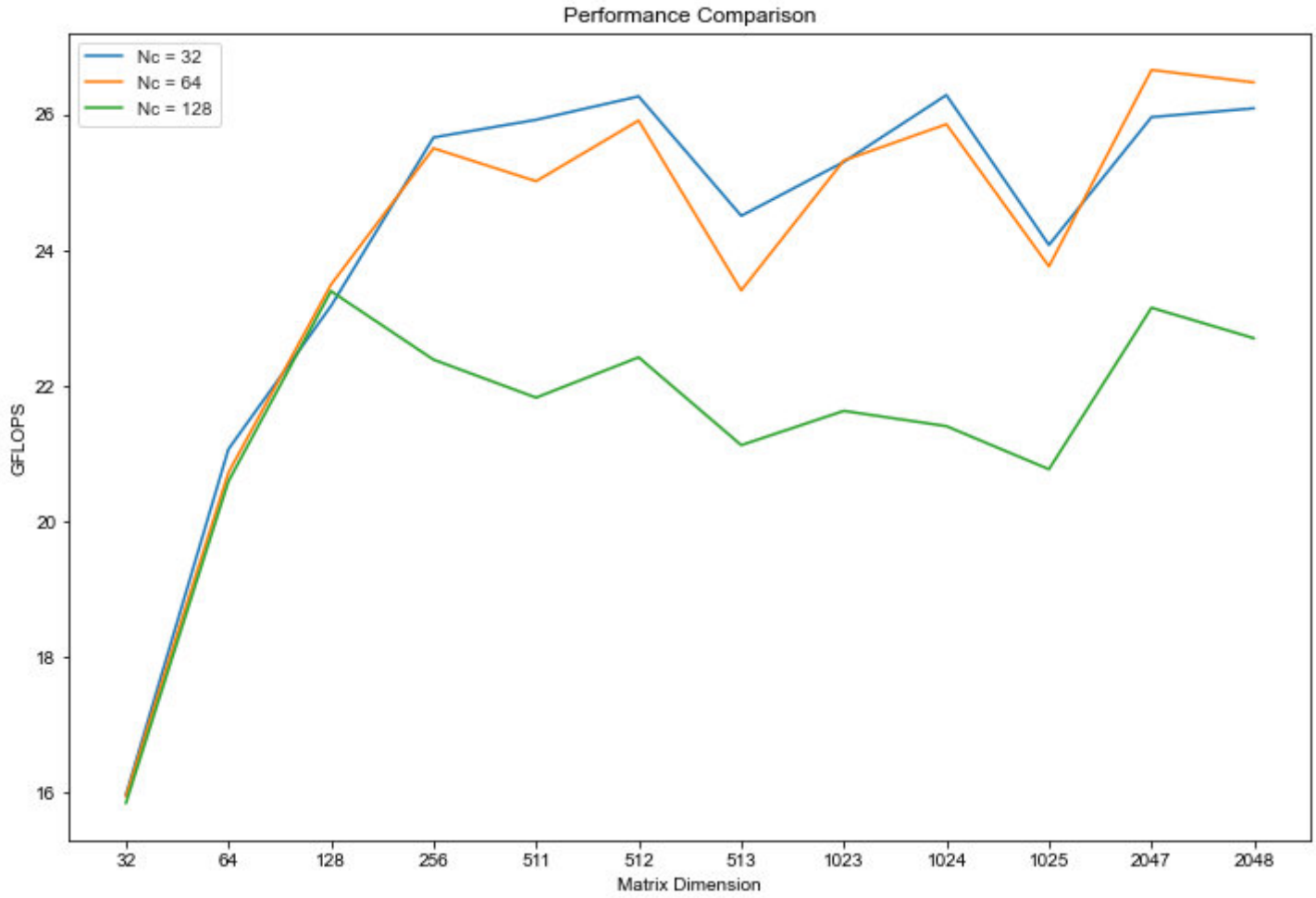
For Mr and Nr, from the result in Q2.b. Development Process, we can see that when $Mr = 4$ and $Nr = 8$, we can make use of 14 registers and get better result.

For Kc, according to the gotoBLAS paper, we should pick Kc as large as possible to amortize the cost of updating the sub-panel of C. However, the $Mr \times Kc$ sub-panel of A should remain in the L1 cache. In addition, considering the set associativity and cache replacement policy, in practice, the $Mr \times Kc$ sub-panel should occupy less than half of the L1 cache so that the elements of A will not be evicted. When $Kc = 512$, it will occupy half of the L1 cache, so we try $Kc = 256, 512, 1024$. The results are shown below.



From the results above, when $Kc = 256$, we can get better result.

For Nc, according to the gotoBlas paper, the $Nc \times Kc$ sub-panel of B should remain in the L2 cache. Also considering the set associativity and cache replacement policy, in practice, the $Nc \times Kc$ sub-panel should occupy less than half of the L2 cache and TLB. So we try $Nc = 32, 64, 128$. The results are shown below.



From the results above, when $Nc = 32$, we can get better result.

For Mc, we find that it have less influence on the result. But $Mc \times Kc$ should remain in L3 cache, so here we set $Mc = 1024$.

Cache Behavior

Here we use `cachegrind` to check the performance of naive method, our method and OpenBLAS with matrix size range from 32 to 1024. The results are shown below:

- naive method

```
==212040==
==212040== I   refs:      59,268,226,029
==212040== I1  misses:            13,571
==212040== LLi misses:             2,159
==212040== I1  miss rate:           0.00%
==212040== LLi miss rate:           0.00%
==212040==
==212040== D   refs:      17,142,870,814  (16,958,508,397 rd   + 184,362,417 wr)
==212040== D1  misses:     7,599,035,357  ( 7,581,532,789 rd   +  17,502,568 wr)
==212040== LLd misses:           282,074  (        15,538 rd   +     266,536 wr)
==212040== D1  miss rate:           44.3% (          44.7%     +         9.5%  )
==212040== LLd miss rate:            0.0% (           0.0%     +         0.1%  )
==212040==
==212040== LL refs:        7,599,048,928  ( 7,581,546,360 rd   +  17,502,568 wr)
==212040== LL misses:            284,233  (        17,697 rd   +     266,536 wr)
==212040== LL miss rate:             0.0% (           0.0%     +         0.1%  )
```

- our method

```
==134618== I   refs:      8,211,934,165
==134618== I1  misses:           18,372
==134618== LLi misses:            2,301
==134618== I1  miss rate:          0.00%
==134618== LLi miss rate:          0.00%
==134618==
==134618== D   refs:      2,813,002,286  (2,615,434,106 rd   + 197,568,180 wr)
==134618== D1  misses:      446,197,451  (  435,049,360 rd   +  11,148,091 wr)
==134618== LLd misses:          334,938  (       16,667 rd   +     318,271 wr)
==134618== D1  miss rate:          15.9% (         16.6%     +         5.6%  )
==134618== LLd miss rate:           0.0% (          0.0%     +         0.2%  )
==134618==
==134618== LL refs:         446,215,823  (  435,067,732 rd   +  11,148,091 wr)
==134618== LL misses:           337,239  (       18,968 rd   +     318,271 wr)
==134618== LL miss rate:            0.0% (          0.0%     +         0.2%  )
```

- OpenBLAS method

```
==189648== I   refs:      7,475,968,506
==189648== I1  misses:           31,410
==189648== LLi misses:            1,426
==189648== I1  miss rate:          0.00%
==189648== LLi miss rate:          0.00%
==189648==
==189648== D   refs:      2,508,739,510  (2,302,677,960 rd   + 206,061,550 wr)
==189648== D1  misses:      211,468,981  (  200,246,139 rd   +  11,222,842 wr)
==189648== LLd misses:          263,649  (          229 rd   +     263,420 wr)
==189648== D1  miss rate:           8.4% (          8.7%     +         5.4%  )
==189648== LLd miss rate:           0.0% (          0.0%     +         0.1%  )
==189648==
==189648== LL refs:         211,500,391  (  200,277,549 rd   +  11,222,842 wr)
==189648== LL misses:           265,075  (        1,655 rd   +     263,420 wr)
==189648== LL miss rate:            0.0% (          0.0%     +         0.1%  )
```

From the results above we can see that the L1 data cache miss rate for naive code is 44.3%, for our code it is 15.9% and for OpenBLAS it is 8.4%. Therefore, we can see that our code is much more cache-friendly than naive code, but less cache-friendly than OpenBLAS.

Different Organization

The start code has two changes to BLISLab tutorial.

- implement row-major matrices for input matrices A, B and the output matrix C
- handle fringe cases where N is not an even multiple of the various blocking sizes.

The second change is simple. We just need to handle some specific siutations. It would cause some extra overhead.

The first change is because the array is row-major ordered in our machine but column-major ordered in the BLISLab tutorial. So we have to change the skeleton code.

```
In our method:
Loop 5: for ic = 0; ic < m; ic += Mc
Loop 4: for pc = 0; pc < k; pc += Kc
Loop 3: for jc = 0; jc < n; jc += Nc
Loop 2: for ir = 0; ir < Mc; ic += Mr
Loop 1: for jr =0; jr < Nc; jr += Nr
Loop 0: for kr=0; kr < Kc; kr++

In BLISLab:
Loop 5: for ic = 0; ic < n; ic += Nc
Loop 4: for pc = 0; pc < k; pc += Kc
Loop 3: for jc = 0; jc < m; jc += Mc
```

```
Loop 2: for ir = 0; ir < Nc; ic += Nr
Loop 1: for jr =0; jr < Mc; jr += Mr
Loop 0: for kr=0; kr < Kc; kr++
```

We think the difference is because the major order. In our method, we want to start from dividing the row of matrix in the loop5 because matrix is row-major order. This my cause high cache hit in the future. It's similar in the loop2. But in the BLISLab, we want to start from traversing column because it's colum-major order. So we need to modify the code depending on the machine.

Q2.e. Future Work

In x86:

1. We may create more avx functions to handle edge size of the matrix. In this way, we don't need to pad with zero, which could reduce the unnecessary computation and increase the performance.
2. If the computer has more than one core, the multiple threading could help us parallelize the process of micro kernel of our method.

In arm:

1. Mobile devices are very popular nowadays. So we belive if we could build the program on arm platform and run it on different devices, such as phones and ipads.

References

[1] Goto, Kazushige, and Robert A. van de Geijn. "Anatomy of high-performance matrix multiplication." ACM Transactions on Mathematical Software (TOMS) 34.3 (2008): 1-25.

[2] Huang, Jianyu, and Robert A. Van de Geijn. "BLISlab: A sandbox for optimizing GEMM." arXiv preprint arXiv:1609.00076 (2016).

[3] https://software.intel.com/sites/landingpage/IntrinsicsGuide

[4] https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions