Project 3: Improved Matrix Multiplication in C

SID	NAME
12212108	Bai Junyan

Introduction

In Project 2, we have compared the performance of matrix multiplication written in C and Java and reach a conclusion that, generally, C outperforms Java due to better memory management, higher load of CPU and enhanced compiler optimization. In this project, we will focus on improving the overall performance of matrix multiplication in C as much as possible and compare our best participant against OpenBLAS (Open Basic Linear Algebra Subprograms).

So as to "win" the battle(though it's unlikely), we shall know detailed message about OpenBLAS --- why is it so fast?

> *Why OpenBlAS is so fast in Matrix Multiplication?*

OpenBLAS, an open-source implementation of the BLAS interface, is highly optimized specially for nearly every common computer architecture by hundreds of devoted engineers, it makes the most out of the nowadays "Computer Memory Hierarchy" and combine skills including Multi-threading, Cache-aware Algorithm, SIMD Instructions, Compiler Optimizations etc.

The core idea of GEMM in OpenBLAS is based on GotoBLAS, of which the concept is clarified in the essay written by KAZUSHIGE GOTO - "Anatomy of High-Performance Matrix": **If all matrix operands are $n * n$ in size, $O(n^3)$ floating point operations are performed with $O(n^2)$ data so that the cost of moving data between memory layers (main memory, the caches, and the registers) can be amortized over many computations.** Detailed explanation is as follow:

Key Ideas

Divide GEMM into smaller sub problem GEBP,GEPB or GEPDOT, by optimizing the lowest level(the smaller block-size matrix) kernel, then high performance can be acquired in other circumstances.

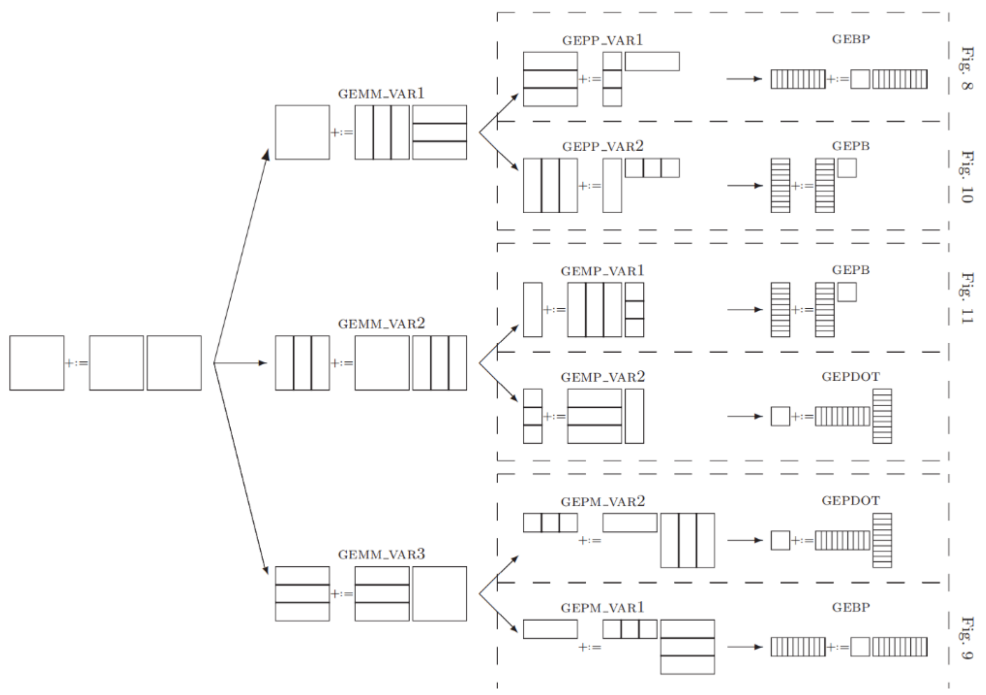


Fig. 4. Layered approach to implementing GEMM.

The pseudocode is as follow:

```
for p = 1:K
    for i = 1:M
        for j = 1:N
            Cij += Aip * Bpj
        endfor
    endfor
endfor
```

To optimize GEPB problems, core idea is to consider the multi-level of Computer memory hierarchy.

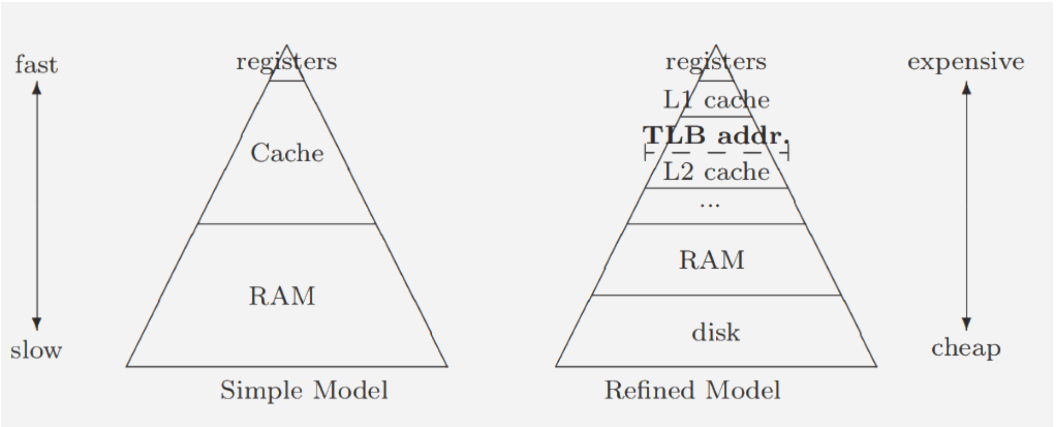


Fig. 6. The hierarchical memories viewed as a pyramid.

Using cache, the pseudocode can be modified to:

```
Load A into cache   (m_c*k_c memops)
for j = 0,.... N-1
    Load B_j into cache (k_c * n_r memops)
    Load C_j into cache (m_c * n_r memops)
    calculate GEPB  (C_j := AB_j + C_j)
    Store C_j into memory (m_c * n_r memops)
endfor
```

By analyzing the concept of **amortizing**, the problem is clear:

- Which level of cache to put matrix A in: as far as possible, to make $m_c * k_c$ as large as possible, as it turns out, L2 cache is our choice. And B_j, C_j are stored into cache. We can measure each R_{comp}, R_{load} as the rate of floating point arithmetic by CPU and loading data from L2 stream to register groups. We need to ensure that

$$n_r \geq \frac{R_{comp}}{2R_{load}}$$

- TLB miss is way more unaffordable than cache miss(cache miss can be compensated by prefetch), need to make sure that:
    - A,B,C can be found by TLB, so that there is no TLB miss when calculating $C_j := AB_j + C_j$
    - A is repeatedly accessed, A should be stay in TLB as long as possible.

Then our solution is:

- Packing: pack A into continuous accessed memory, then choose argument m_c, k_c to fulfill the above requirement. Also, packing B or C is better.
- Store all of the matrix by column major order, which would make the multiplication $C_{aux} := A^{-}B_j$ consecutive in memory.
- Choosing $m_r * n_r$
- Choosing k_c: To amortize, k_c should be as large as possible.
    - B_j' $k_c n_r$ floating point number should only take up below half the space of L1 cache.
    - $m_c k_c$(the space of A^{-}) floating point number should take up much of L_2.

My Implementation

Based on the approach proposed by GOTO and following the critical steps of another essay - "BLISlab: A Sandbox for Optimizing GEMM" (BLIS is based on OpenBLAS and performs nearly as good as it), I conduct my experiment in the following steps:

- Implementing plain matrix multiplication and plain SIMD optimization first, and compare their performance.
- Implementing blocked plain matrix multiplication and compare the performance difference.
- Implementing the scheme for GEPB and test its performance.
- Optimizing for the $8 * 4$ `micro-bench` and test its performance shift.
- Go parallelizing utilizing `OpenMP`.
- Testing performance on the server.

Note that some famous algorithm such as Strassen are not implemented since they are "theoretically" fast, which is in practice not ideally fast especially for GEMM in rather smaller dimension. I mainly focus using Computer itself for the most.

Experiment

Local Experiment Environment and Project Structure

Hardware Environment:

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          20
On-line CPU(s) list:             0-19
Thread(s) per core:              2
Core(s) per socket:              10
Socket(s):                       1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           154
Model name:                      12th Gen Intel(R) Core(TM) i7-12700H
Stepping:                        3
CPU MHz:                         2688.011
BogoMIPS:                        5376.02
Virtualization:                  VT-x
Hypervisor vendor:               Microsoft
Virtualization type:             full
L1d cache:                       480 KiB
L1i cache:                       320 KiB
L2 cache:                        12.5 MiB
L3 cache:                        24 MiB
```

Software Environment:

```
GNU Make 4.2.1
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)
```

The project structure is as follow, `dclock.c` is a time measuring function adapted from the original `bl2_clock()` routine in the BLIS library.

```
├── dclock.c
├── m.c
├── main.c
├── makefile
├── output.t
└── parameter.h
```

Step0: Plain Matrix Multiplication

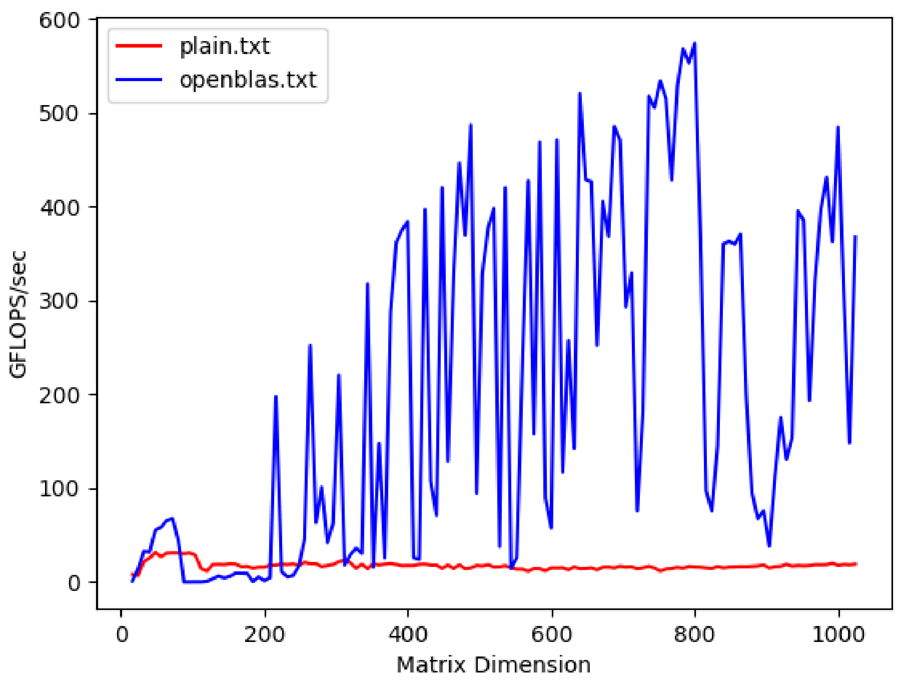
In step0, plain matrix multiplication is implemented as follow, note that **macro is defined to make sure matrices are stored in column-major**. The linear array at address C is used to store elements $C_{i,j}$ so that i, j elements is mapped to location `j * ldc + i`.

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

`a,b,c` are matrices `m * k, k * n, m * n` each, while all of them are embedded in a larger array that has each `lda, ldb, ldc` rows, which are called "leading dimension".

```
void matmul_plain(int m,int n,int k,float *a,int lda,float *b,int ldb,float *c,int ldc){
    if(a==NULL||b==NULL||c==NULL) return;
    int i,j,p;
    for(i=0;i<n;i++){
        for(j=0;j<k;j++){
            for(p=0;p<m;p++){
                C(p,i) = C(p,i) + A(p,j)*B(j,i);
            }
        }
    }
}
```

Compile this with `-O3 -march=native` and compare the GFLOPS with OpenBLAS library with largest matrix dimension as 1024, the result is as follow:



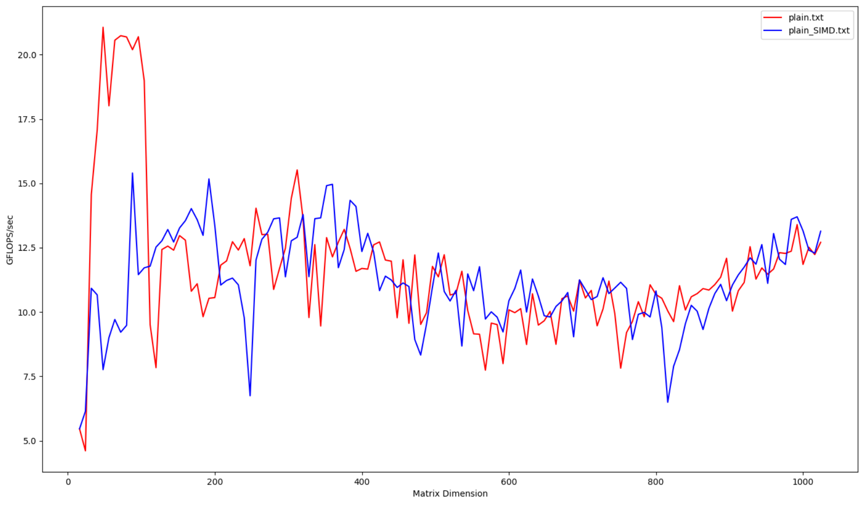
OpenBLAS literally kills the plain implementation. Let's take a closer look at the plain GEMM. With the increase of the matrix size, GLOPS drop significantly with plain implementation. Since the order of loop has already been changed, cache miss is significantly reduced due to continuous memory access.

Step0.5: Plain SIMD

Now's let's use plain SIMD to optimize the code, since my CPU is Intel's supporting `AVX2`, then 256-bit register `__m256` can be used in GEMM which can calculate 8 elements of `c` within one iteration. The key point here is the use of `_mm256_set1_ps` which store `B(j,i)` to all elements of y, avoiding storing data into register in the inner `p` loop repeatedly (in the inner loop, `B(j,i),B(j+1,i),...,B(j+7,i)` all needs to be loaded into register).

```
void matmul_plain_SIMD(int m,int n,int k,float *a,int lda,float *b,int ldb,float *c,int ldc){
    if(a==NULL||b==NULL||c==NULL) return;
    if(m%8!=0||n%8!=0||k%8!=0) fprintf(stderr,"Input size must be multiplier of 8!");
    int i,j,p;
    __m256 x,y;
    __m256 gamma=_mm256_setzero_ps();
    for(i=0;i<n;i++){
        for(j=0;j<k;j++){
            y=_mm256_set1_ps(B(j,i)); //store B[i,j] to all element of
y(the jth column)
            for(p=0;p<m;p+=8){
                x=_mm256_loadu_ps(&A(p,j));
                gamma=_mm256_add_ps(gamma,_mm256_mul_ps(x,y));
                _mm256_storeu_ps(&C(p,i),gamma);
            }
        }
    }
}
```

The performance comparison between `matmul_plain` and `matmul_plain_SIMD` is as follow:



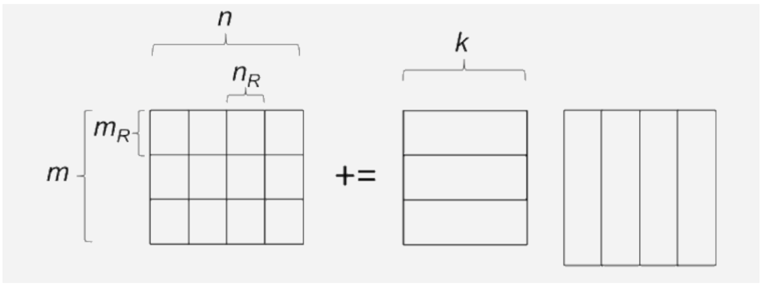
Surprisingly, the overall performance between two of them are nearly the same with `matmul_plain` outperforms `matmul_plain_SIMD` at smaller dimensions. The problem is located at the `-O3` compiler optimization, in the assembly code, instruction with prefix `-v` indicates that SIMD is used:

```
.L75:
    vmovups (%rax,%rdi), %xmm3
    vmovups (%r12,%rdi), %xmm4
    vinsertf128 $0x1, 16(%rax,%rdi), %ymm3, %ymm0
    vinsertf128 $0x1, 16(%r12,%rdi), %ymm4, %ymm1
    vfmadd132ps %ymm2, %ymm1, %ymm0
    vmovups %xmm0, (%r12,%rdi)
    vextractf128    $0x1, %ymm0, 16(%r12,%rdi)
    addq    $32, %rdi
    cmpq    %r13, %rdi
    jne .L75
    cmpq    %r14, %r15
```

After manually implementing SIMD instructions, the compiler cannot grasp what the program is essentially doing, leading to performance decreasing.

Step1: GEPB Prepared

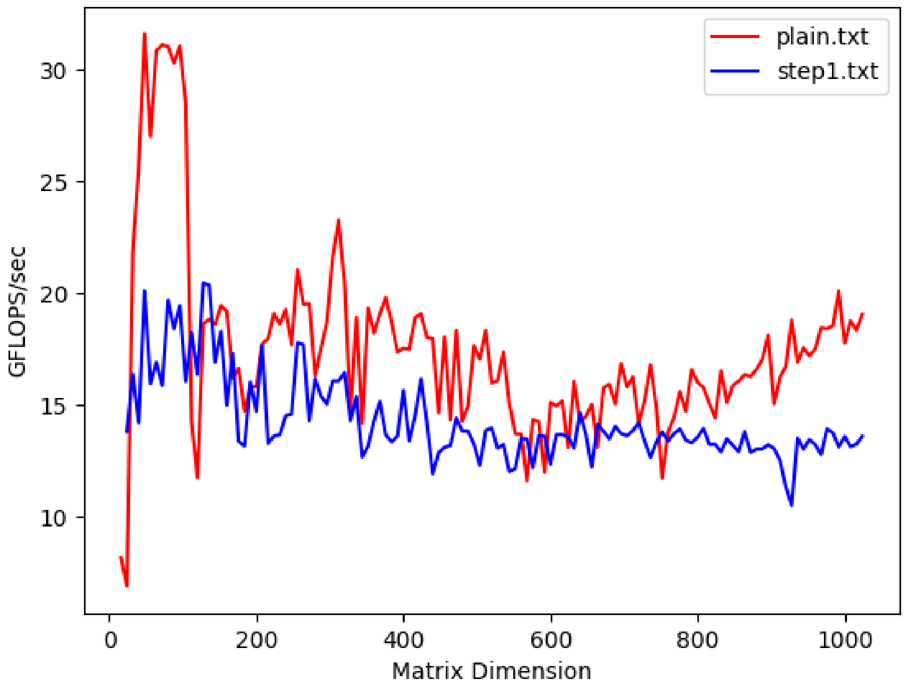
From Goto's essay, it's essential to optimize GEPB on `micro kernal`. In step1, our "modest first goal" is to utilize this idea and conduct GEMM under GEPB. Essentially, what we want is to implement GEPB so that $m_R * n_R$ blocks of C are kept in registers.



Implementation is as follow, two functions `AddDot` and `AddMM` are defined to calculate matrix multiplication for each $m_R * n_R$ blocks. And fundamental optimizations are implemented, such as loop unrolling and pointer points to the top of the column. `DGEMM_NRr` and `DGEMM_MRr` are predefined as macro which are equal to m_R and n_R, they are set to 4 for convenience at first.

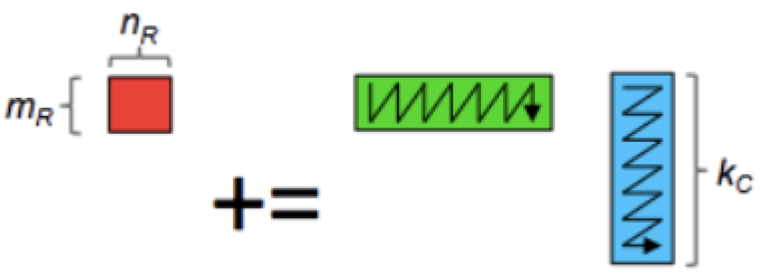
```c
void matmul_tech(int m,int n,int k,float *a,int lda,float *b,int ldb,float *c,int ldc){
    if(a==NULL||b==NULL||c==NULL) return;
    int i,j;
    for(j=0;j<n;j+=DGEMM_NRr){
        for(i=0;i<m;i+=DGEMM_MRr){
            AddMM(k,&A(i,0),lda,&B(0,j),ldb,&C(i,j),ldc);
        }
    }
}
void AddDot(int k,float *a,int lda,float *b,int ldb,float *gamma){
    if(a==NULL||b==NULL||gamma==NULL) return;
    int p;
    for(p=0;p<k;p++){
        *gamma += A(0,p)*B(p,0);//A's row multiply by B's column
    }
}
void AddMM(int k,float *a,int lda,float *b,int ldb,float *c,int ldc){
    if(a==NULL||b==NULL||c==NULL) return;
    int h,l;
    for(h=0;h<DGEMM_NRr;h++){
        for(l=0;l<DGEMM_MRr;l++){
            AddDot(k,&A(l,0),lda,&B(0,h),ldb,&C(l,h));
            }
        }
    }
```

However, with `-O3 march=native` option turned on, there is essentially no difference between `matmul_plain` and `matmul_tech` (former one outperforms the latter one) which entails the fact again that how much optimization the compiler has already done and what we do is only messing up with it. But we are moving towards the right direction for further optimization.

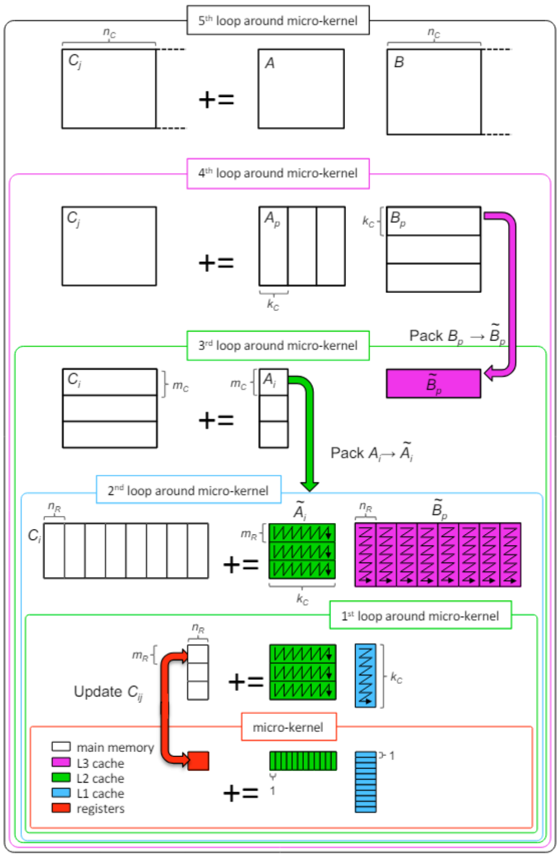


Step2: GEPB Blocking

Now it's time to implement the approach of GEPB, which is blocking for multiple levels of cache. To amortize the cost of memory loading on computing of floating point, the bottom `micro-kernal` should be highly optimized. And in order to get the best performance, it helps if all data is accessed contiguously, which is why at some point prior to reaching the micro-kernel, data is packed in the order indicated by the arrows:



In the Intro section, it has been discussed that since each column of the block of A in the above picture is multiplied by each element in the corresponding row of the block of B (which are so-called `micro-panels`), the latency to the L2 cache (the time required to bring in an element of the micro-panel of A from that cache) can be amortized over $2n_R$ flops. And by `prefetching` techniques (slightly different from `prefetching` in computer memory system, but the idea is the same), we can put continuously accessed columns on each of $A's$ `micro-panel` onto registers to amortize even further. And the `micro-kernal` of B are kept in L1 cache, which we have to choose parameter k_c carefully. Ideally, this approach may reach the so-called `locally optimzal`, in the sense that assuming data is in a certain memory layer in the hierarchy, the proposed blocking at that level optimally amortizes the cost of data movement with the next memory layer.


5th loop around micro-kernel
n_C
C_j
+=
A
B
4th loop around micro-kernel
C_j
A_p
k_C
B_p
Pack $B_p \rightarrow \tilde{B}_p$
3rd loop around micro-kernel
C_i
m_C
A_i
$\tilde{B}_p$
Pack $A_i \rightarrow \tilde{A}_i$
2nd loop around micro-kernel
n_R
$\tilde{A}_i$
m_R
$\tilde{B}_p$
1st loop around micro-kernel
Update C_{ij}
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

Currently, the whole scheme for this approach is established, so the only remaining task to do is to optimize for the calculation of in `micro-kernal`, which would tell exactly the computer to store A into L2 cache, B into L3 cache, continuous A into registers and B_j into L1 cache, just as we want. There are two ways to complete this task: use `avx2` in C or directly write x86 assembly code. Though in Computer Organization class we learn about RISC-V, I'm not so famailiar with X86. I will use `avx2` to write optimize for the `micro-kernal`. Note that the code is adapted from `double` implementation by "BLIS Framwork".

Step2.1: GEPB Scheme

In this step, we will leave the optimization for `micro-kernal` aside temporarily and complete the Scheme for GEPB proposed in the paper. Critically, there are several factors we need to take into consideration.

- The parameters `DGEMM_NC`, `DGEMM_KC` ,`DGEMM_MC`, `DGEMM_MRr` and `DGEMM_NRr` which each stands for n_c,k_c,m_c, m_r and n_r in the above graph. As what we have discussed in **Step2.1**, these parameters represent the blocksize splited on the way to `micro-kernal`, and all of them have some restrictions, considering a single core on my CPU:
    - $k_c * n_c$ packed B should be kept in L3 cache, which is `24MiB`.
    - $m_c * k_c$ packed A should be kept in L2 cache for most of the time, which is `1.25MiB`.
    - $n_R * k_c$ `micro-panels` B_j should be kept in L1 cache, which is `48KiB`.

We first take $m_R * n_R$ as $8 * 4$ which means 32 single floating points number, each SIMD register is 256-bit, then we need four registers to store them.(This is the size for `micro-kernal`). Since in Goto's paper, experiment shows that B_j should only take up less than half of L1 cache to ensure it would stay there when calculating, thus k_c should be set to 256. Then we set m_c to be 640 to make sure A take more than half of the L2 cache. Finally, n_c can be set up to 24567 since the shared L3 cache is very large, we will set it to 8192.

To pack A, B into continuous space, we can simply set a temporary pointer pointing to the discrete space in A and B, like `ptr = A(i+di,j+dj)`, however, the discontinuous space will cause a lot of trouble, we could inform the compiler to optimize it by use pointers. Also, we declare the two packing functions to be `inline` function. Notice the slight difference between the order data is scanned in A and B in `micro-panel`.

```c
inline void packA(int m,int k,float *x,int ldx,int offset,float *pack){//data should be accessed by column first
    if(x==NULL||pack==NULL) return;
    int i,p;
    float *a_ptr[DGEMM_MRr];
    for(i=0;i<m;i++){a_ptr[i]=x+(offset+i);}
    for(i=m;i<DGEMM_MRr;i++){a_ptr[i]=x+(offset+0);}
    for(p=0;p<k;p++){
        for(i=0;i<DGEMM_MRr;i++){
            *pack = *a_ptr[i];
            pack ++;
            a_ptr[i] = a_ptr[i]+ldx;
        }
    }
}
inline void packB(int n,int k,float *y,int ldy,int offset,float *pack)
{//data should be accessed by row first
    if(y==NULL||pack==NULL) return;
    int j,p;
    float *b_ptr[DGEMM_NRr];
    for(j=0;j<n;j++){b_ptr[j]=y+ldy*(offset+j);}
    for(j=n;j<DGEMM_NRr;j++){b_ptr[j]=y+ldy*(offset+0);}
    for(p=0;p<k;p++){
        for(j=0;j<DGEMM_NRr;j++){
            *pack++ = *b_ptr[j]++;
        }
    }
}
```

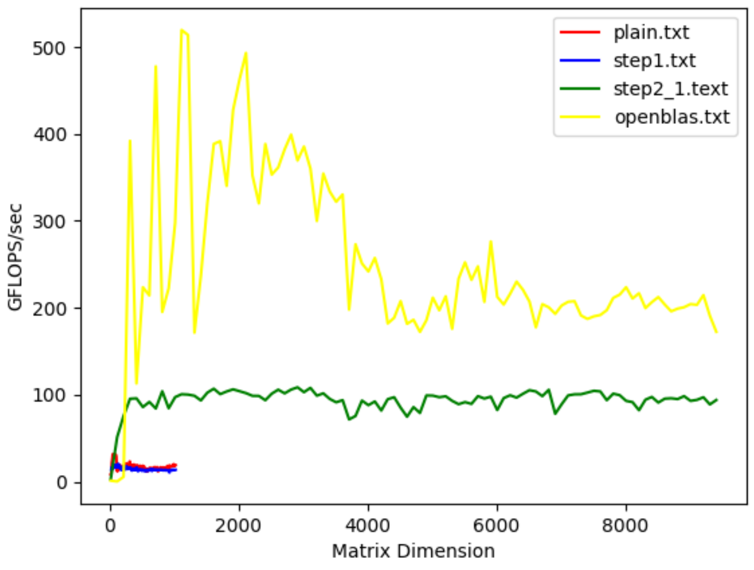
When allocating space for the packed datas, aligned memory is allocated to optimize for `micro-kernal`.

```
float *malloc_aligned(int m,int n,int size){
    float *ptr;
    int e;
    e = posix_memalign((void**)&ptr,(int)32,size*m*n);//needs a pointer
to the pointer to reallocate the memory
    if(e){
        printf("aligned memory aligned fails");
        exit(1);
    }
    return ptr;
}
```

Then it's time to write the 6 loops to approach GEPB, we add the previous `packA` and `packB` each to Loop3 and Loop4 outside of the `micro-kernal`, one thing to emphasize is while entering the next loop, we need to check whether our preset `DEMM_Mx` is out of boarder of the matrix size. We always choose the smaller one by calling `pb=min(k-pc,DGEMM_KC)`.

```
for(jc=0;jc<n;jc+=DGEMM_NC){//Loop5
        jb=min(n-jc,DGEMM_NC);//ensure that the remaining doesn't exceed
the columns of B when packing
        for(pc=0;pc<k;pc+=DGEMM_KC){//Loop4
            pb=min(k-pc,DGEMM_KC);
            for(j=0;j<jb;j+=DGEMM_NRr){//PackB,iterate through column of
B
                packB(min(jb-
j,DGEMM_NRr),pb,&b[pc],ldb,jc+j,&_B_[j*pb]);
            }
            for(ic=0;ic<m;ic+=DGEMM_MC){//Loop3
                ib=min(m-ic,DGEMM_MC);
                for(i=0;i<ib;i+=DGEMM_MRr){//PackA,iterate through row
of A
                    packA(min(ib-
i,DGEMM_MRr),pb,&a[pc*lda],m,ic+i,&_A_[i*pb]);
                }
                for(jr=0;jr<jb;jr+=DGEMM_NRr){//Loop2
                    jrb=min(jb-jr,DGEMM_NRr);
                    for(ir=0;ir<ib;ir+=DGEMM_MRr){//Loop1
                        irb=min(ib-ir,DGEMM_MRr);
                        for(kr=0;kr<pb;kr++){//Loop0--micro-kernal
                            C(ir,jr) = A(ir,kr)*B(kr,jr) + C(ir,jr);
                        }
                    }
                }
            }
        }
    }
```

The result is as follow:



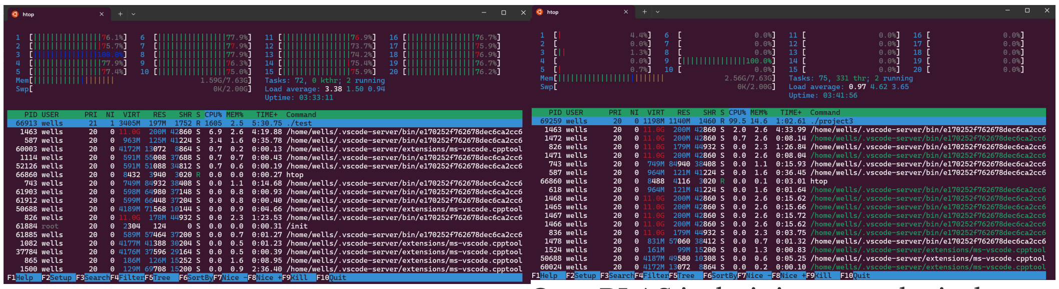
It is unsurprising to see that when the matrix size is small, let's say, smaller than $3000 * 3000$, OpenBLAS still way outperforms the GEPB. That's because, well, remember when setting the essential parameter, they are set as follow:

```c
#define LDA 10000
#define LDB 10000
#define LDC 10000
#define DGEMM_MRr 8
#define DGEMM_NRr 4
#define DGEMM_NC 8192
#define DGEMM_MC 640
#define DGEMM_KC 256
```

m_c, n_c, k_c are set to a rather larger value to optimize for caches operation(modern computer has larger and larger cache), and when the matrix size is small, GEPB method is not actually used since we choose the smaller parameter when entering the inner loop.

Still, large performance shift can be seen (more than 10 times faster). And when the matrix size is large, around $8000 * 8000$, we are getting closer to the result of OpenBLAS! That's approximately half of performance, and what's better is that the GFLOPS remain rather stable even the size is getting large.

Before optimizing for the `micro-kernal` manually, let's dive deeper why OpenBLAS is faster than the current implementation. Using `htop` on Shell, we can get (OpenBLAS on the left):



```
  1  [||||||||||||||||76.1%]   6  [||||||||||||||||77.9%]   11 [||||||||||||||||76.9%]   16 [||||||||||||||||76.7%]
  2  [||||||||||||||||75.7%]   7  [||||||||||||||||77.9%]   12 [||||||||||||||||73.7%]   17 [||||||||||||||||75.9%]
  3  [||||||||||||||||100.0%]  8  [||||||||||||||||77.9%]   13 [||||||||||||||||74.2%]   18 [||||||||||||||||76.9%]
  4  [||||||||||||||||77.9%]   9  [||||||||||||||||76.3%]   14 [||||||||||||||||75.4%]   19 [||||||||||||||||76.7%]
  5  [||||||||||||||||77.4%]   10 [||||||||||||||||75.0%]   15 [||||||||||||||||75.9%]   20 [||||||||||||||||76.2%]
  Mem[|||||||||||||||||||          1.59G/7.63G]   Tasks: 72, 0 kthr; 2 running
  Swp[                               0K/2.00G]   Load average: 3.38 1.50 0.94
                                                  Uptime: 03:33:11

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  66913 wells       21   1 3405M  197M  1752 R 1605  2.5  5:30.75 ./test
   1463 wells       20   0 11.0G  200M 42860 S  6.9  2.6  4:19.88 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
    587 wells       20   0  963M  125M 41224 S  3.4  1.6  0:35.78 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
  60003 wells       20   0 4172M 13072  8864 S  0.7  0.2  0:00.13 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
   1114 wells       20   0  591M 55008 37688 S  0.7  0.7  0:00.43 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
  52126 wells       20   0  591M 51088 34812 S  0.7  0.6  0:00.19 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
  66860 wells       20   0  8432  3940  3020 R  0.0  0.0  0:00.27 htop
    743 wells       20   0  749M 84932 38408 S  0.0  1.1  0:14.68 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
  61903 wells       20   0  598M 64980 37148 S  0.0  0.8  0:00.93 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
  61912 wells       20   0  599M 66448 37204 S  0.0  0.8  0:00.40 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
  50688 wells       20   0 4189M 71568 10144 S  0.0  0.9  0:04.66 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
    826 wells       20   0 11.0G  178M 44932 S  0.0  2.3  1:23.53 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
  61884 root        20   0  2304   124     0 S  0.0  0.0  0:00.31 /init
  61885 wells       20   0  589M 57464 37200 S  0.0  0.7  0:01.27 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1082 wells       20   0 4177M 41388 30204 S  0.0  0.5  0:01.23 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
  37784 wells       20   0 4176M 37596 29164 S  0.0  0.5  0:00.39 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
    865 wells       20   0  186M  126M 15252 S  0.0  1.6  0:08.95 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
   1500 wells       20   0  129M 69708 15200 S  0.0  0.9  2:36.40 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

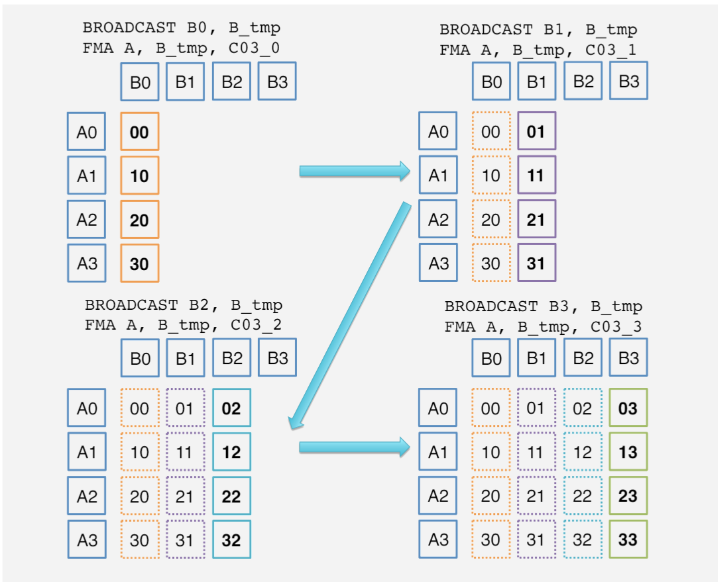
```
  1  [|               4.4%]   6  [                0.0%]   11 [                0.0%]   16 [                0.0%]
  2  [                0.0%]   7  [                0.0%]   12 [                0.0%]   17 [                0.0%]
  3  [||              1.3%]   8  [                0.0%]   13 [                0.0%]   18 [                0.0%]
  4  [                0.0%]   9  [|||||||||||||||100.0%]   14 [                0.0%]   19 [                0.0%]
  5  [|               0.7%]   10 [                0.0%]   15 [                0.0%]   20 [                0.0%]
  Mem[||||||||||||||||||||||||     2.56G/7.63G]   Tasks: 75, 331 thr; 2 running
  Swp[                               0K/2.00G]   Load average: 0.97 4.62 3.65
                                                  Uptime: 03:41:56

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  69259 wells       20   0 1198M 1140M  1460 R 99.5 14.6  1:02.61 ./project3
   1463 wells       20   0 11.0G  200M 42860 S  2.0  2.6  4:33.99 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1472 wells       20   0 11.0G  200M 42860 S  0.7  2.6  0:08.14 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
    826 wells       20   0 11.0G  179M 44932 S  0.0  2.3  1:26.84 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1471 wells       20   0 11.0G  200M 42860 S  0.0  2.6  0:08.04 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
    743 wells       20   0  749M 84940 38408 S  0.0  1.1  0:15.93 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
    587 wells       20   0  964M  121M 41224 S  0.0  1.6  0:36.45 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
  66860 wells       20   0  8488  4116  3020 R  0.0  0.1  0:03.01 htop
    618 wells       20   0  964M  121M 41224 S  0.0  1.6  0:01.64 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1468 wells       20   0 11.0G  200M 42860 S  0.0  2.6  0:15.62 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1465 wells       20   0 11.0G  200M 42860 S  0.0  2.6  0:15.66 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1467 wells       20   0 11.0G  200M 42860 S  0.0  2.6  0:15.72 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1466 wells       20   0 11.0G  200M 42860 S  0.0  2.6  0:15.62 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
    836 wells       20   0 11.0G  179M 44932 S  0.0  2.3  0:03.75 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1478 wells       20   0  831M 57060 38412 S  0.0  0.7  0:01.32 /home/wells/.vscode-server/bin/e170252f762678dec6ca2cc6
   1524 wells       20   0  161M   99M 15200 S  0.0  1.3  0:00.83 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
  50688 wells       20   0 4187M 49580 10308 S  0.0  0.6  0:05.25 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
  60024 wells       20   0 4172M 13072  8864 S  0.0  0.2  0:00.10 /home/wells/.vscode-server/extensions/ms-vscode.cpptool
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

OpenBLAS is draining every logical cores of CPU, thus multithreading technology is used, while my plain GEPB implementation only use one single core, that explains the difference in performance. Let's move on to optimize the floating point operation on `micro-kernal`.

Step2.2: `Micro-Kernal` Optimization

Let's implement a simpler `micro-kernal` using `avx2` first based on the simple `matmul_plain_SIMD` we implemented in step0. Simpler means we plainly use `avx2` for quicker floating point arithmetic but not optimizing for cache use. The implementation can be abstracted to this graph:



Because my CPU doesn't support `__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c)`, I can only use `_mm256_add_ps` and `_mm256_mul_ps` combined, which will increase the cost of computation.

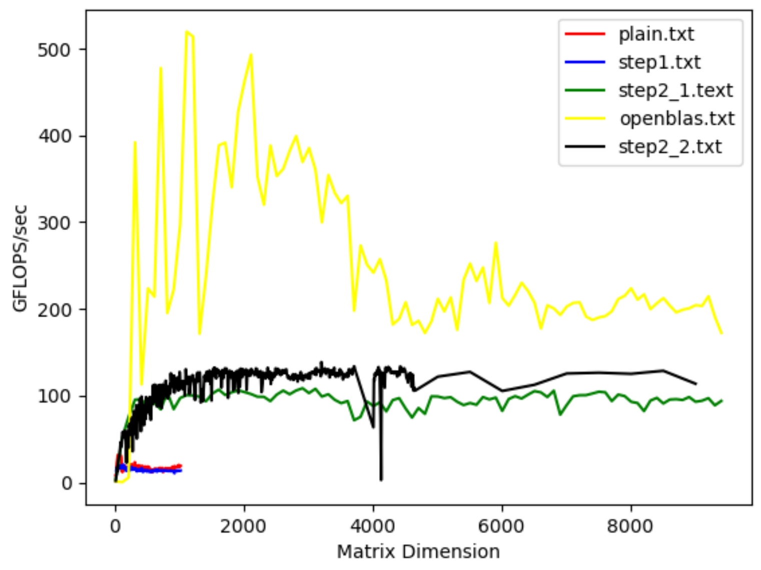
```
for (int p = 0; p < k; p++) {
        __m256 a_0p_a_1p_vreg = _mm256_load_ps(a);
        __m256 a_2p_a_3p_vreg = _mm256_load_ps(a + 2);
        a += 4;

        __m256 b_p0_vreg = _mm256_broadcast_ss(b);
        __m256 b_p1_vreg = _mm256_broadcast_ss(b + 1);
        __m256 b_p2_vreg = _mm256_broadcast_ss(b + 2);
        __m256 b_p3_vreg = _mm256_broadcast_ss(b + 3);
        b += 4;
        c_00_c_10_vreg = _mm256_add_ps(c_00_c_10_vreg,
_mm256_mul_ps(a_0p_a_1p_vreg, b_p0_vreg));
        c_01_c_11_vreg = _mm256_add_ps(c_01_c_11_vreg,
_mm256_mul_ps(a_0p_a_1p_vreg, b_p1_vreg));
        c_02_c_12_vreg = _mm256_add_ps(c_02_c_12_vreg,
_mm256_mul_ps(a_0p_a_1p_vreg, b_p2_vreg));
        c_03_c_13_vreg = _mm256_add_ps(c_03_c_13_vreg,
_mm256_mul_ps(a_0p_a_1p_vreg, b_p3_vreg));

        c_20_c_30_vreg = _mm256_add_ps(c_20_c_30_vreg,
_mm256_mul_ps(a_2p_a_3p_vreg, b_p0_vreg));
        c_21_c_31_vreg = _mm256_add_ps(c_21_c_31_vreg,
_mm256_mul_ps(a_2p_a_3p_vreg, b_p1_vreg));
        c_22_c_32_vreg = _mm256_add_ps(c_22_c_32_vreg,
_mm256_mul_ps(a_2p_a_3p_vreg, b_p2_vreg));
        c_23_c_33_vreg = _mm256_add_ps(c_23_c_33_vreg,
_mm256_mul_ps(a_2p_a_3p_vreg, b_p3_vreg));

    }
```

And the result is as follow:



We can see that after slight optimization on `micro-kernal`, the performance is getting closer to OpenBLAS, reaching nearly 64% of its performance. One thing to notice is that my there are actually two modes of my computer -- "Quiet Mode" and "Game Mode",which differs in CPU Clock Frequency. In this case, I only use the "Quiet Mode" for reference. But when doing parallelizing, CPU Clock Frequency matters a lot,(So OpenBLAS's performance right now it's demined), we will see it in step3.

Step3: Parallelizing with `OpenMP`

Since there are 20 logical cores in my CPU, parallelizing is definitely approachable taken that there are multiple embedded loops. In this project, for convenience and thread safety concern, I choose to use `OpenMP` library to go parallelizing. The question is, which loop to parallelize. By reading another paper "Anatomy of High-Performance Many-Threaded Matrix Multiplication", we can get some insights.

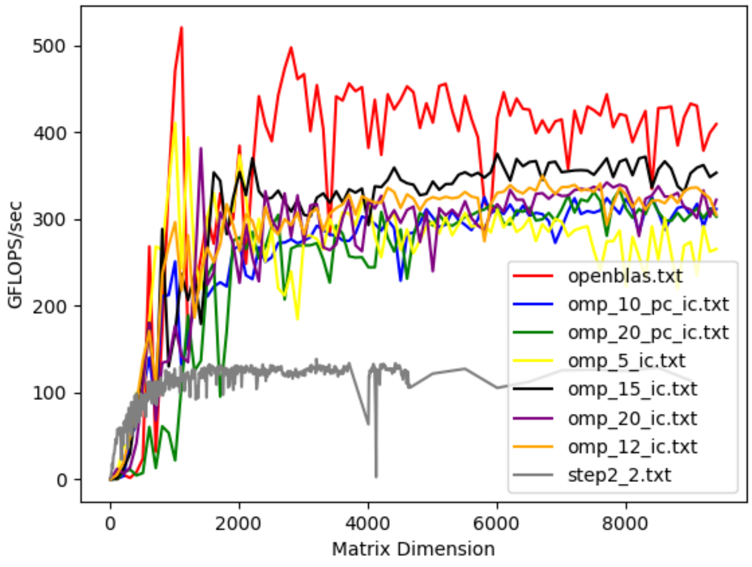
- Parallelizing within Loop0: This is parallelizing when calculating the `micro-kernal`, which is not suggested. There are two factors:
  - We need multiple thread to calculate a single block of C, increasing the cost of the `reduce` operations of threads.
  - The calculation is just too little for every thread since the matrix size is rather small within Loop0, `amortation` is not implemented.
- Parallelizing within Loop1: This is parallelizing in loop `ir`, which is calculating every `micro-panel` of A multiply by the same column B_j which is in L1 cache. To amortize the cost of loading B_j from L3, factor $\frac{mc}{mr}$ must be large for us to parallelize.

- Parallelizing within Loop2: This is parallelizing within loop `jr`, which is calculating every `micro-panel` of B multiply by the same row of A_i that is stored in L2 cache. To amortize the cost of packing the shared A_i on every core, factor $\frac{n_c}{n_r}$ must be large for us to parallelize.
- Parallelizing within Loop3: This is parallelizing within loop `ic`, which is splitting the m rows of A by m_c. So we need $\frac{m}{m_c}$ large, which is satisfied since we set m to 640.
- Parallelizing within Loop4: This is parallelizing within loop `pc`, which is calculating the exact $C(ir, jr)$. Since each thread need to update for it, thread competence exits, will likely lead to poor performance.
- Parallelizing within Loop5: This is parallelizing within loop `jc`, which is parallelizing for the whole matrix multiplication. This involves shared L3 cache fetch, which is better for multi socket machine with `NUMA`.

We will experiment parallelizing within Loop3, Loop4 and 5. To achieve assigning nearly the same workload for each thread, we need to separate the task to fit in different threads, we define an inline function called `thread_shared`:

```
inline void thread_shared(int n,int bf,int *start,int *end){
    int way =omp_get_num_threads();th
    int id = omp_get_thread_num();
    //thread partitioning
    //we partition the space between all_start and
    // all_end into n_way partitions, each a multiple of block_factor
    // with the exception of the one partition that recieves the
    // "edge" case (if applicable).
    int all_start=0;
    int all_end=n;
    int size=all_end-all_start;
    int whole=size/bf;
    int left=size%bf;
    int lo=whole/way;int hi=whole/way;
    //differentiate between edge cases
    int b_th_lo=whole%way;
    if(lo!=0) lo++;
    int size_lo=lo*bf,size_hi=hi*bf;
    // Precompute the starting indices of the low and high groups.
    int lo_start=all_start,hi_start=all_start+b_th_lo*size_lo;
    // Compute the start and end of individual threads' ranges
    // as a function of their work_ids and also the group to which
    // they belong (low or high).
    if(id<b_th_lo){
        *start=lo_start+id*size_lo;
        *end=lo_start+(id+1)*size_lo;
    }
    else{*start=hi_start+(id-b_th_lo)*size_hi;
    *end=hi_start+(id-b_th_lo+1)*size_hi;
    if(id==way-1) *end+=left;}
}
```

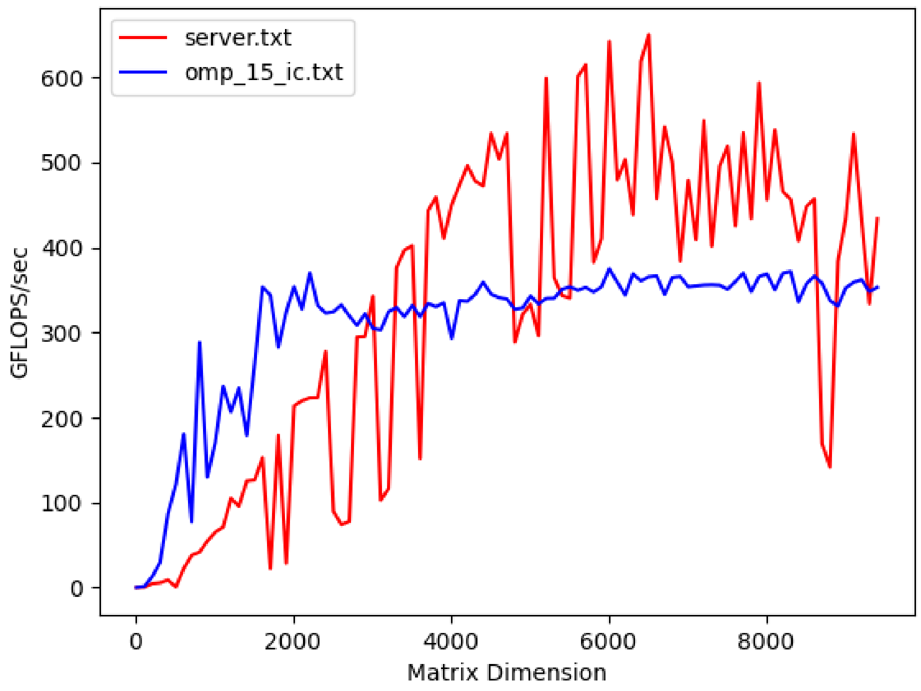
Then experiment parallelizing within different loops, the result compared to OpenBLAS is as follow. One thing to note is that `Game Mode` of my computer is turned on to enable higher CPU frequency. It can be seen that both OpenBLAS and my implementation increases):



Compared with single thread, GFLOPS is around 2.5 times bigger. And right now we are reaching more than 75% the performance of OpenBLAS! And the result shows that only parallelizing within Loop3 has the best performance among them. Adding parallelizing within Loop2(`pc`) is indeed worsen the performance, in correspondence to the previous theory.

Step4: Testing on the Server

Testing result on the server, we can get



As expected, it outperfoms local performance since the better CPU and more cores.

Running the $64000 * 64000$ matrix multiplication, the result is as follow:

My	Server
3.451305e+02 GFLOPS/sec	4.8999e+02 GFLOPS/sec

Conclusion

By applying the method proposed by Goto's paper, I reach nearly 75% of the performance of OpenBLAS at last. This project really makes me realize the importance of diving deep into the computer memory and optimize our code according to it. I find it amazing when this project reflects close connection to what I've learned in another course Computer Organization, which encourages me to learn more about the bottom level knowledge. For example, I read the famous article "What every programmer should know about computer memory", which inspires me a lot.

There are still things I want to do but fail to accomplish, and I wish to complete them in the future:

- optimize the `micro-bench` by writing assembly code, taking advantage of the cache memory.
- Using GPU to run the code and see its performance.
- Fully understand how MKL and OpenBLAS works so well on different platforms by learning the source code.

Reference

1. https://github.com/OpenMathLib/OpenBLAS/wiki
2. https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf
3. https://www.cs.utexas.edu/~flame/pubs/blis3_ipdps14.pdf
4. https://github.com/flame/blislab
5. https://github.com/flame/how-to-optimize-gemm