Matrix Multiplication

Intro

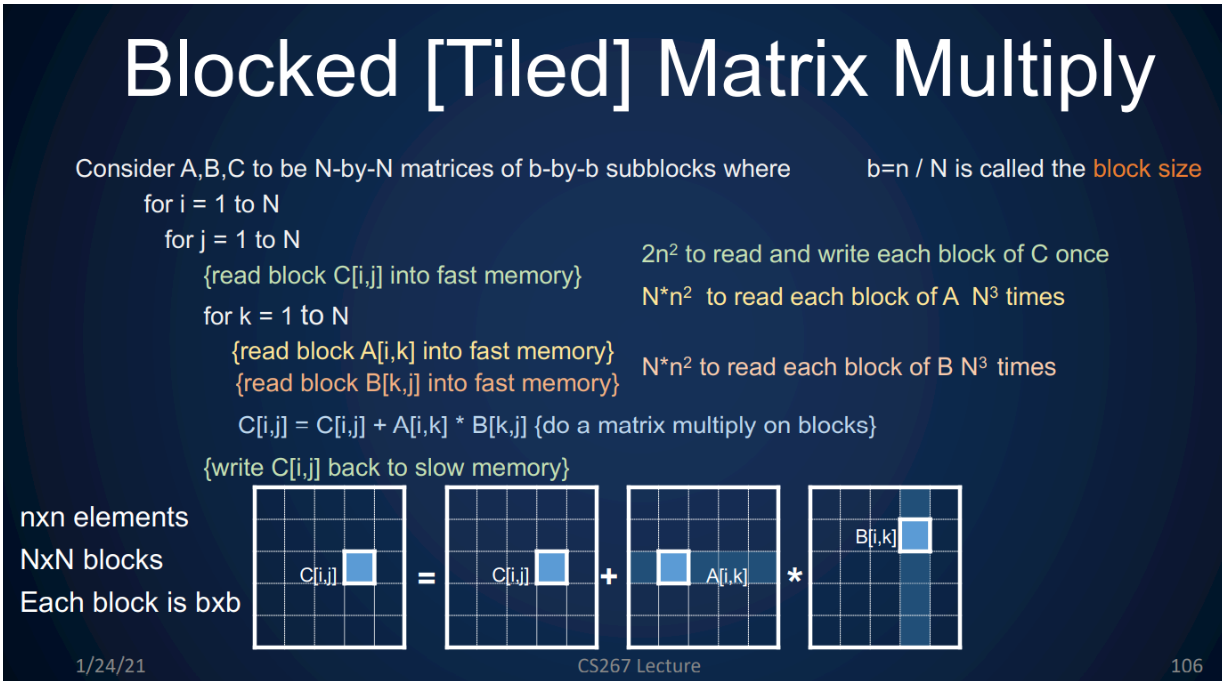
As we all know, Matrix Multiplication(MM) is widely used in many area, such as Computer Graphics . Deep Learning, etc. In these project, we use 9 ways to achieve Matrix Multiplication $A_{n\times n} \times B_{n\times n} = C_{n\times n}$. In `dgemm1` and `dgemm2`, I use 1 and 12 **register** to accelerate the Multiplication. Especially, in `dgemm2`, we use 2*2 size of block multiplication to reduce running time. Theoretically, naive computational intensity(CI) is $q = f/m = 2n^3/(n^3 + 3n^2) \approx 2$, while block is $q = 2n^3/((2N+2) \times n^2) \approx n/N = b$, which b means block size. To explore the improvement of registers, `dgemm4` without registers is fairly compared to `dgemm2`.

Besides, matrix wise MM, which means block size can be arbitrary, is implemented in `dgemm5`, to make is easier to measure, we set the block size `B=4`. However, the size of cache greatly affect the **best** block size `B`. In theory, if cache size is M_c , it must satisfy $3b^3 < M_c$. So actually, in real machine, there must have some problems when computing and need to set block size manually, which may not operate at its best. Thus, **Cache-oblivious** method is needed, which means you don't need to know M_c for this to work. The computational intensity is $CI = 2n^3/O(n^3/\sqrt{M}) = O(\sqrt{M})$. And to achieve this goal, **recursive** method is used, so we call these ways "recursive".

Additionally, considering **locality**, in recursive way, we must *divide and rule*. So if the matrix size is too big to fit the cache, the access of data can be a huge cost. Thus, in $A \times B = C$, we reorder the data on A, B to **Z-morton**, so that it can fit the cache and improve our performance.

Idea

Blocked MM



(1) a

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

b

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

##	##	##	##	##	##	##	##
##	##	##	##	##	##	##	##
##	##	##	##	##	##	##	##
##	##	##	##	##	##	##	##
##	##	##	##	##	##	##	##
##	##	##	##	##	##	##	##
##	##	##	##	##	##	##	##
##	##	##	##	##	##	##	##

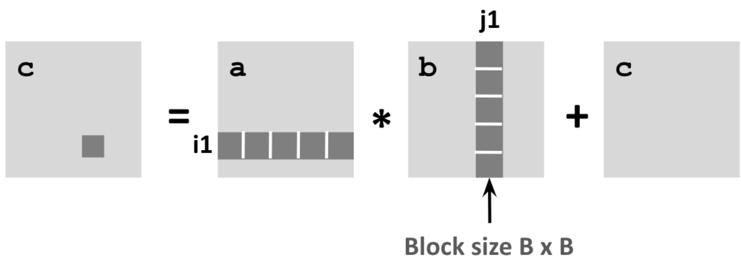
- Theoretically, naive computational intensity(CI) is $q = f/m = 2n^3/(n^3 + 3n^2) \approx 2$, and block CI is $q = 2n^3/((2N + 2) \times n^2) \approx n/N = b$. If $b > 2$, it's more efficient.
- Must satisfy $3b^3 < M_c$

Block Wise

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
    for (j = 0; j < n; j+=B)
             for (k = 0; k < n; k+=B)
         /* B x B mini matrix multiplications */
                  for (i1 = i; i1 < i+B; i++)
                      for (j1 = j; j1 < j+B; j++)
                          for (k1 = k; k1 < k+B; k++)
                          c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



- To achieve block wise MM, we need inner loop.
  - May reduce the efficient.

Recursive

According to the Linear Algebra:

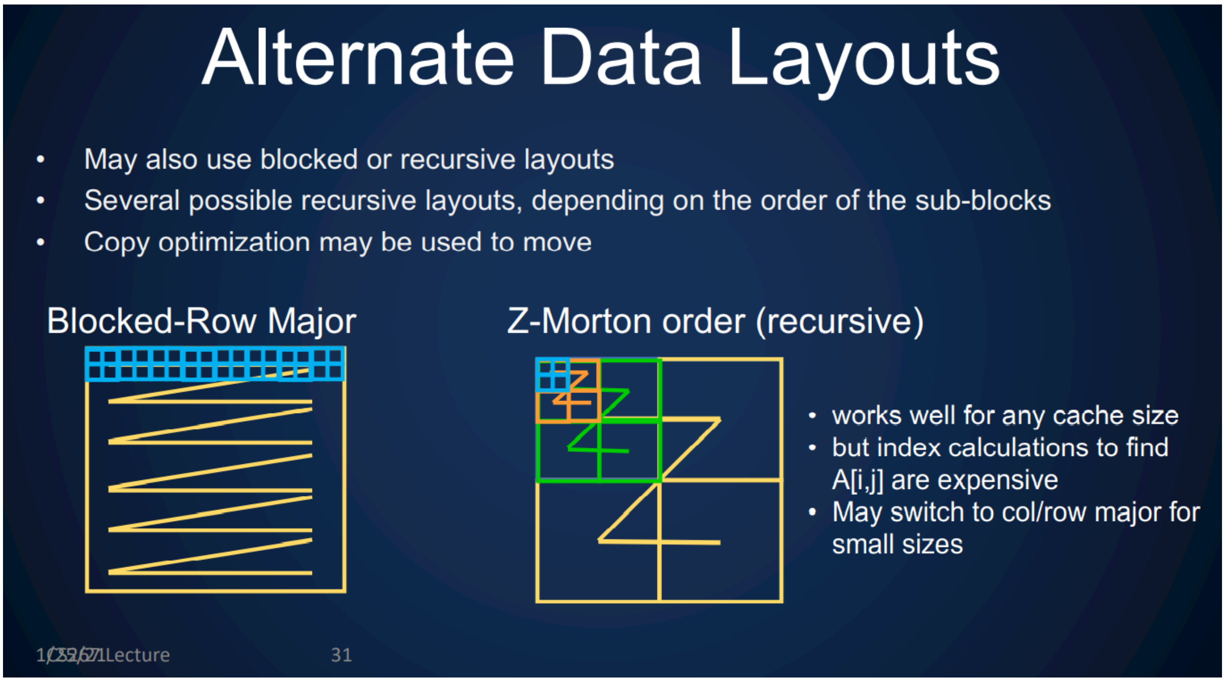
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

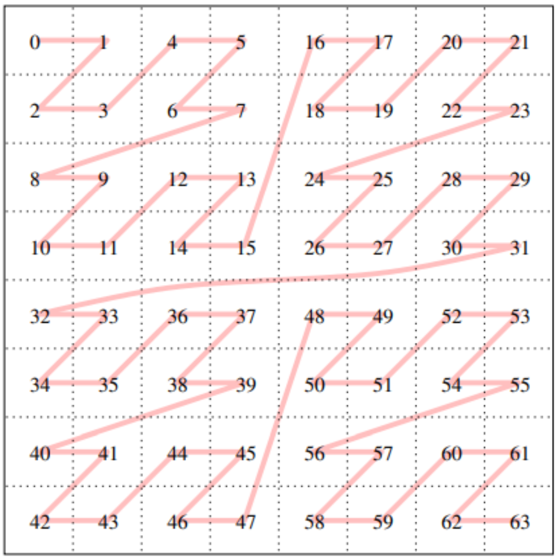
So, we have these code:

```
Define C = RMM (A, B, n)
if (n==1) {
    C00 = A00 * B00 ;
} else{
    C00 = RMM (A00 , B00 , n/2) + RMM (A01 , B10 , n/2)
    C01 = RMM (A00 , B01 , n/2) + RMM (A01 , B11 , n/2)
    C10 = RMM (A10 , B00 , n/2) + RMM (A11 , B10 , n/2)
    C11 = RMM (A10 , B01 , n/2) + RMM (A11 , B11 , n/2)
}
return C
```

- We analyze the CI and get the result: $CI = 2n^3/O(n^3/\sqrt{M}) = O(\sqrt{M})$.
  - This method didn't need cache size M_c, and it will fit the cache automatically.

Z-morton



- Considering **locality**, in recursive way, we must *divide and rule*. So if the matrix size is too big to fit the cache, the access of data can be a huge cost.
  - We reorder the data on A, B to **Z-morton**, which shows like below.
  - 

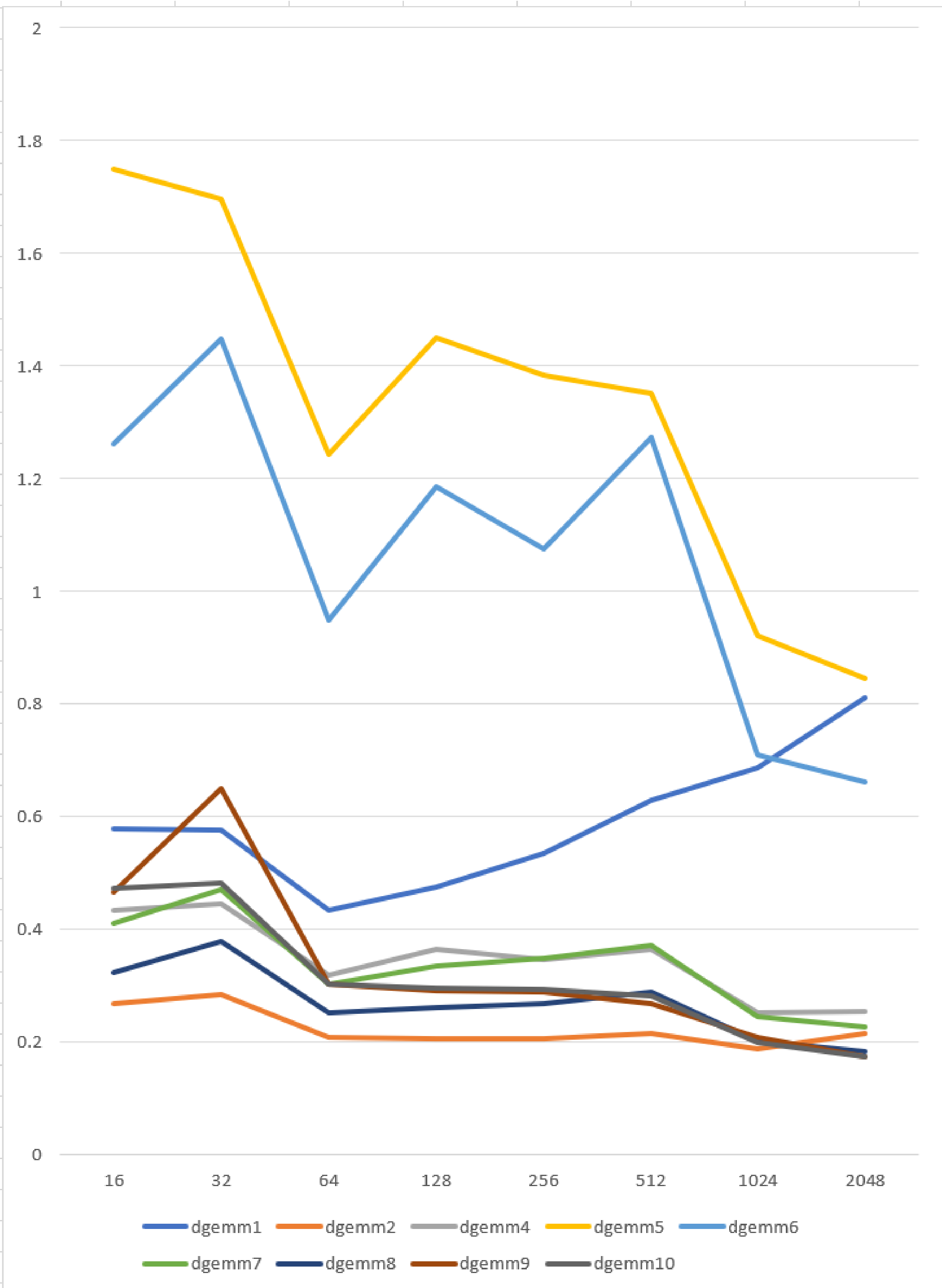
- (The number in the cell is the index order in memory.)
- Generate `z_index` before compute: improve speed at the cost of space.

Result analyze

- At the first line `dgemm0`, the data is the running time (nanoseconds).
- At the following line `dgemm2,4-10`, the data is the ratio of running time, comparing to `dgemm0`.

matrix size	**Remark**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**Avg**	**Note**
dgemm0	Standard	1.80E+04	1.20E+05	1.06E+06	7.92E+06	6.14E+07	5.52E+08	6.49E+09	5.62E+10		Running time
dgemm1	1 register	0.5778	0.5769	0.4326	0.4755	0.5350	0.6277	0.6871	0.8118	0.5905	Ratio to dgemm0
dgemm2	2*2 block +12*reg	**0.2667**	**0.2843**	**0.2072**	**0.2061**	**0.2052**	**0.2138**	**0.1865**	0.2159	0.2232	
dgemm4	2*2 (B=2)	0.4333	0.4447	0.3186	0.3647	0.3468	0.3641	0.2510	0.2545	0.3472	
dgemm5	matrix wise B=4	1.7500	1.6966	1.2433	1.4510	1.3825	1.3520	0.9201	0.8443	1.3300	
dgemm6	**Recursive**	1.2611	1.4489	0.9475	1.1851	1.0745	1.2724	0.7083	0.6602	1.0698	
dgemm7	Rec+2*2	0.4111	0.4713	0.3027	0.3341	0.3486	0.3711	0.2455	0.2276	0.3390	
dgemm8	Rec+2*2+reg	0.3222	0.3791	0.2524	0.2611	0.2675	0.2885	0.2012	0.1832	0.2694	
dgemm9	Rec+2*2+Z	0.4667	0.6500	0.3016	0.2914	0.2884	0.2667	0.2086	0.1747	0.3310	
dgemm10	Rec+2*2+reg+Z	0.4722	0.4821	0.3019	0.2962	0.2929	0.2819	0.1992	**0.1743**	0.3126	

- The line chart of `dgemm2,4-10` :



- Compare and Analyze
  1. `dgemm0` and `dgemm1`
     - Register improves a lot.
     - Save 50% time.
  2. `dgemm0` and `dgemm4`
     - 2*2 block improves a lot.
     - Save $60\% \sim 70\%$ time.
  3. `dgemm2` and `dgemm4`
     - both 2*2 block
     - Save 36% time.

        - Register improves a lot.
    4. `dgemm5` and `dgemm6`
        - without reg
        - `B=4`: Save 20% time.
        - Recursive methods improves
    5. `dgemm6` and `dgemm7`
        - 2*2 block improve a lot: **0.84->0.30**
        - Save 65% time.
    6. `dgemm7` and `dgemm8`
        - Register improves a little
    7. `dgemm8` and `dgemm9`
        - Z-morton's improve is **better** than Register
        - As matrix size improving, Disk -> Mem is more important than Mem -> Reg
    8. `dgemm9` and `dgemm10`
        - Z-morton with Register improves a little
        - Maybe as matrix size improving, some of the reg applications will be failed.
        - Frequent in and out stack slow down the speed.
- Total
    - As matrix size improving, in size of 16-1024, `degmm2` with 2*2 block+reg is the fastest.
    - However, at `n=2048` and more, recursive is better than `degmm2`.

Problem

- When implementing Recursive method, I use `memcpy()` in the function.
    - These causes many data access and slow down the computing.
- When implementing Z-morton method, I use `2Ddecode_z()` in the function.
    - These causes index transfer each time, and has negative effect for performance.
- **Evaluation**
    - To get best performance, we should:
        - use more index to compute
        - less data transfer
        - notice locality
        - improve speed at the cost of space

Conclusion and Discussion

These projects achieve **Block Wise, Recursive, Z-morton** methods of MM. And the best method's running time improves 83% compared to standard MM. Besides, more other methods can be used to improve this MM task: openmp(parallel, simd), and Strassen.