Name: Chen-yang Yu
Student ID: 862052273
1.Register Reuse
Part #1
Since the CPU time =

$$CPI(Cycle\ per\ Instruction) \times (Instruction\ Count) \times (Clock\ time) = \frac{CPI \times (Instruction\ Count)}{frequency}$$

The time for finishing ***dgemm0*** (when n=1000):

$$\frac{(\frac{2}{4}+400)\times n^3}{2Ghz} = \frac{(\frac{2}{4}+400)\times 1000^{\ 3}}{2\times 10^{\ 6}} = 200.25\ (sec)$$

Each in-est loop (k-loop) has 2 floating-point operations(addition and multiplication) and 4 extra memory access (load a, load b,load c, and save c).

The time for wasting on non-register:

$$\frac{400\times n^3}{2Ghz} = \frac{400\times 1000^{\ 3}}{2\times 10^{\ 6}} = 200\ (sec)$$

The time for finishing ***dgemm1*** (when n=1000):

$$\frac{(\frac{2}{4}+200)\times n^3+200\times n^2}{2Ghz} = \frac{(\frac{2}{4}+200)\times 1000^3+200\times 1000^2}{2\times 10^6} = 100.35\ (sec)$$

Each in-est loop (k-loop) has 2 floating-point operations(addition and multiplication) and 2 extra memory access (load a, laod b). Each middle loop (j-loop) has 2 extra memory access (load c, save c).

The time for wasting on non-register:

$$\frac{200\times n^3+200\times n^2}{2Ghz} = \frac{200\times 1000^3+200\times 1000^2}{2\times 10^6} = 100.1\ (sec)$$

When it comes to calculating Gflops, there are 2 floating-point operations(addition and multiplication) in the in-est loop. Multiply 2 with iteration count (n^3):

The performance of ***dgemm0:*** $[\ (\ 2 \times n^3) \ / \ t\ ] \ / \ 1,000,000,000$ (Gflops)

The performance of ***dgemm1:*** $[\ (\ 2 \times n^3) \ / \ t\ ] \ / \ 1,000,000,000$ (Gflops)

The performance evaluation and comparison is in Part#2 (with ***dgemm2***)

Part #2
There are 16 floating-point operations(8 multiplication of a and b, 4 addition of a*b, 4 addition to c) in the in-est loop. Multiply 2 with iteration count ((n/2)^3):

The performance of ***dgemm2:*** $[\ (\ 16 \times (\frac{n}{2})^3) \ / \ t\ ] \ / \ 1,000,000,000$ (Gflops)

With n = 64, 128, 256, 1024, 2048, the time consumption on algorithm ***dgemm0, dgemm1*** and ***dgemm2*** are listed below. ***dgemm2*** has the best performance of all case.

(dgemm_2_1_0_Result.o220049)

n	64	128	256	512	1024	2048
dgemm0	3.322000	29.474001	265.161987	2851.11792	25120.1289	701097.750
dgemm1	2.284000	20.389999	163.266998	2081.76196	17574.5937	451838.625
dgemm2	0.841000	6.727000	53.576000	728.062012	6969.10888	178313.687

(millisecond)

n	64	128	256	512	1024	2048
dgemm0	0.15782299	0.14232448	0.12654314	0.089478	0.085899	0.024508
dgemm1	0.229548	0.205704	0.205519	0.128946	0.122193	0.038022

dgemm2*	0.623410	0.623503	0.626296	0.368699	0.308143	0.096346

(Gflops)

Part #3

Choose block size = 3. Use 9 register in the middle loop to hold the value of C from memory load. Use 3 register for first column of matrix A and 3 register for first row of matrix B in the in-est loop (k-loop). Replace them with the next column/ row to calculate the sum.

There are 54 operation in the in-est loop, so the performance Gflops is 54 multiply with iteration count (n/3)^3

The performance of ***dgemm2:*** $[\,(\,54 \times (\frac{n}{3})^3)\,/\,t\,]\,/\,1,000,000,000$ (Gflops)

In order to compare the performance of ***dgemm0, dgemm1, dgemm2*** and ***dgemm3*** , we choose n = 64, 128, 256, 1024, 2048 as matrix size. However, there will be some element not able to use the algorithm we set when facing the boundary issue. For example, If the matrix size is not the multiple of 2, the rest one element on the boundary cannot use the ***dgemm2*** to solve. In order to fully compare all of the algorithm without consider other variables, we make a some adjustments in n.

With n = 64, 128, 256, 1024, 2048, the time consumption on algorithm ***dgemm0, dgemm1, dgemm2*** and ***dgemm3*** are listed below. ***dgemm3*** has the best performance of all case.

(dgemm_3_2_1_0_Result.o220048)

n	64	126	252	510	1020	2046
dgemm0	2.529000	23.398001	217.600006	1685.82299	17836.8105	241644.546
dgemm1	1.545000	16.090000	136.285995	1126.31396	11036.7265	208374.578
dgemm2	0.606000	6.305000	53.805000	449.518005	5108.87890	96560.0156
dgemm3	0.460000	4.514000	41.065979	327.458008	3157.10888	21873.6054

(millisec)

n	64	126	252	510	1020	2046
dgemm0	0.20731039	0.17098692	0.14708647	0.132651	0.117912	0.071077
dgemm1	0.279612	0.248648	0.234845	0.235549	0.192305	0.082206
dgemm2	0.712871	0.634536	0.594852	0.590192	0.415437	0.177398
dgemm3*	0.939130	0.886299	0.779380	0.810186	0.672266	0.783117

(Gflops)

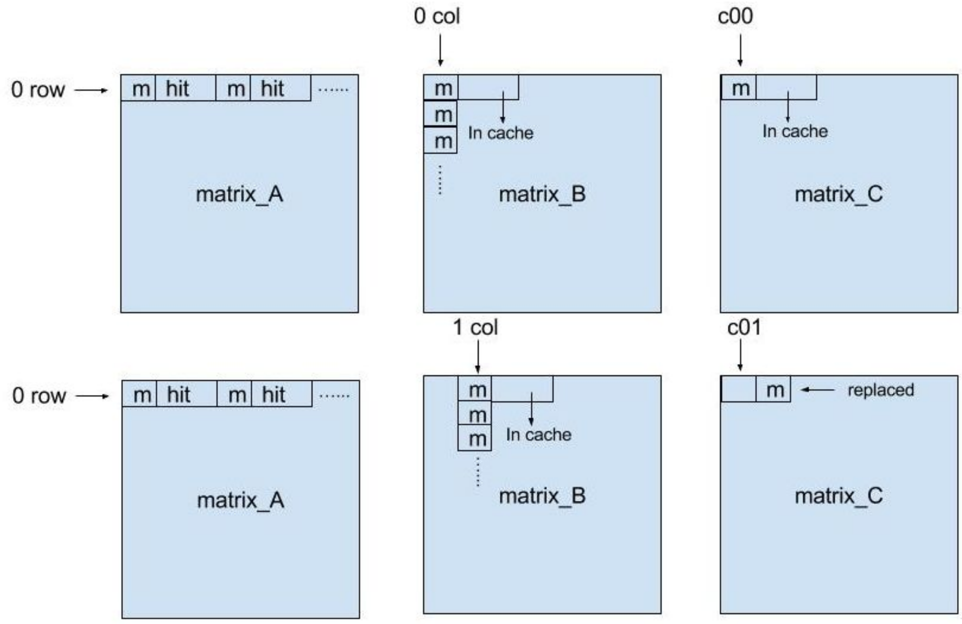
2.Cache Reuse

Part #1

10000x10000

Since the cache has only 60 lines, it is impossible to hit cache if we read the element column by column from memory(the space is not enough to hold 10000 cache line). When we access the first element of the next column, the data ought to be in cache has already be replaced. We may hit the rest 9 element after first cache miss is we read the element row by row. However, it is still impossible to hit a cache when we access this element again.

Algorithm ijk(jik):



The first iteration show that matrix_A miss 1 time out of 10. Matrix_B and matrix_C are all miss. Although the row 0 of matrix_A was in cache, row 0 has already replaced at the time we read it again (when the second iteration). As the result, the whole matrix_A miss 1 time out of 10. Cache hits when column%10 is not 0. Matrix_B and matrix_C would all miss since at least 10000 times cache miss has been made while there is only 60 lines of cache.

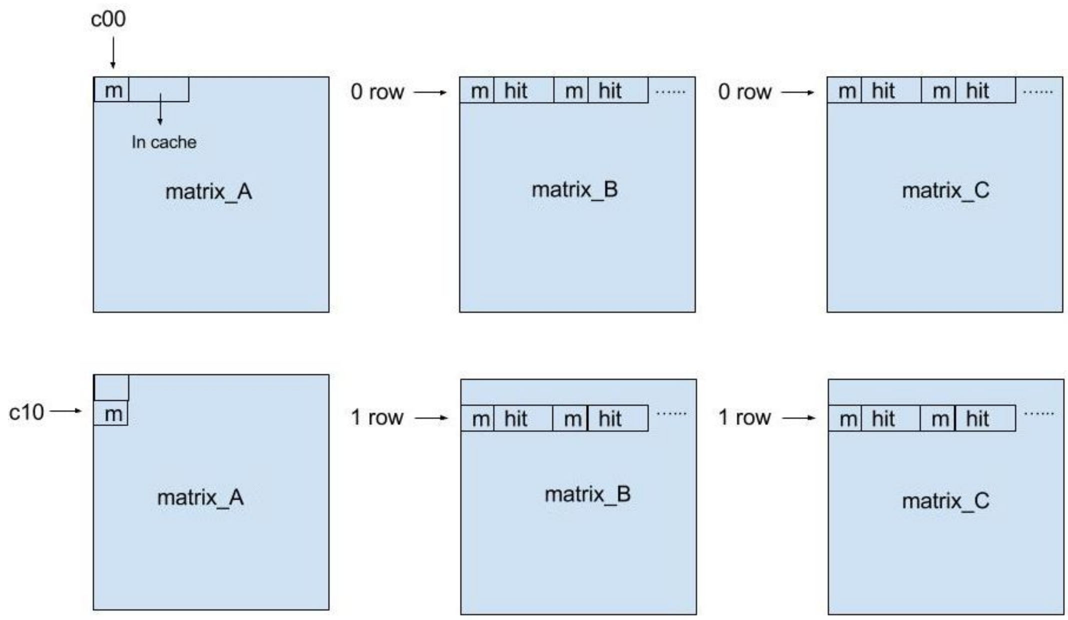
As the result:

Read cache miss rate of Matrix_A = (1000x10000x10000)/(10000x10000x10000) = 1/10

Read cache miss rate of Matrix_B = (10000x10000x10000)/(10000x10000x10000) = 1

Read cache miss rate of Matrix_C = (10000x10000x10000)/(10000x10000x10000) = 1

Algorithm kij(ikj):



Matrix_A amd matrix_B cache miss 1 time out of 10 in the first iteration while matrix_C is cache miss. Matrix_A and matrix_B still cache miss even when access the same row (being replaced)in the rest of iteration. Cache hits when column%10 is not 0. Cache all miss for matrix_C(no matter kij or ikj).

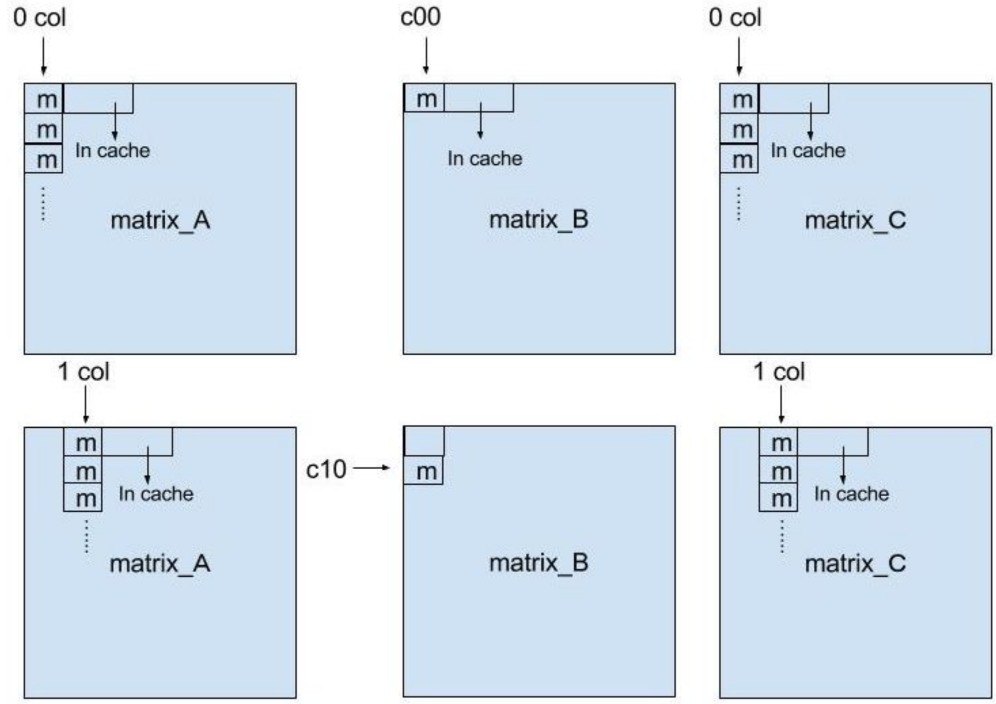
As the result:

Read cache miss rate of Matrix_A = (10000x10000x10000)/(10000x10000x10000) = 1

Read cache miss rate of Matrix_B = (1000x10000x10000)/(10000x10000x10000) = 1/10

Read cache miss rate of Matrix_C = (1000x10000x10000)/(10000x10000x10000) = 1/10

Algorithm jki(jki):

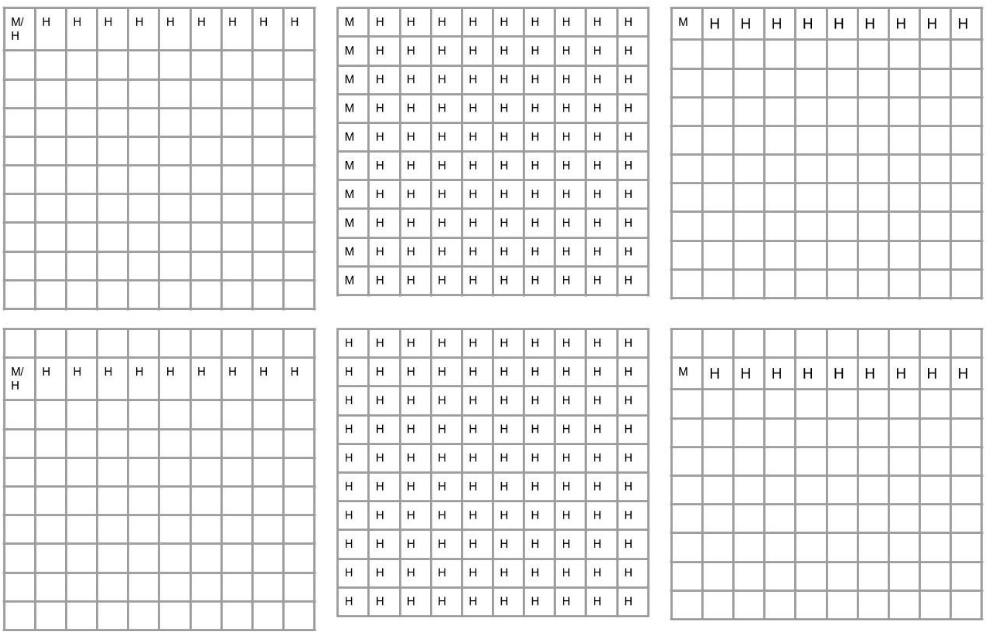


Matrix_A, matrix_B and matrix_C are all miss since cache space is not enough to hold the data till next time access.
As the result:
Read cache miss rate of Matrix_A = (10000x10000x10000)/(10000x10000x10000) = 1
Read cache miss rate of Matrix_B = (10000x10000x10000)/(10000x10000x10000) = 1
Read cache miss rate of Matrix_C = (10000x10000x10000)/(10000x10000x10000) = 1

10*10

There are only totally 3x10 line in the sum of Matrix_A, Matrix_B, and Matrix_C. We can obtain all of the element value in the cache after the cache miss of first time access the row.

Algorithm ijk(jik):



When it comes to the first middle-loop (calculate the first row of matrix_C), matrix_A miss the first element and 9 hit of the rest. Matrix_B miss all 10 element. Matrix_C miss. The next k-loop, matrix_A, matrix_B, and matrix_C all hits. Matris_A and matrix_Cis the same in the second middle-loop while matrix_B all hits.

The first element of matrix_A, matrix_B, and matrix_C miss whenever first time access. Hit when the other time access.

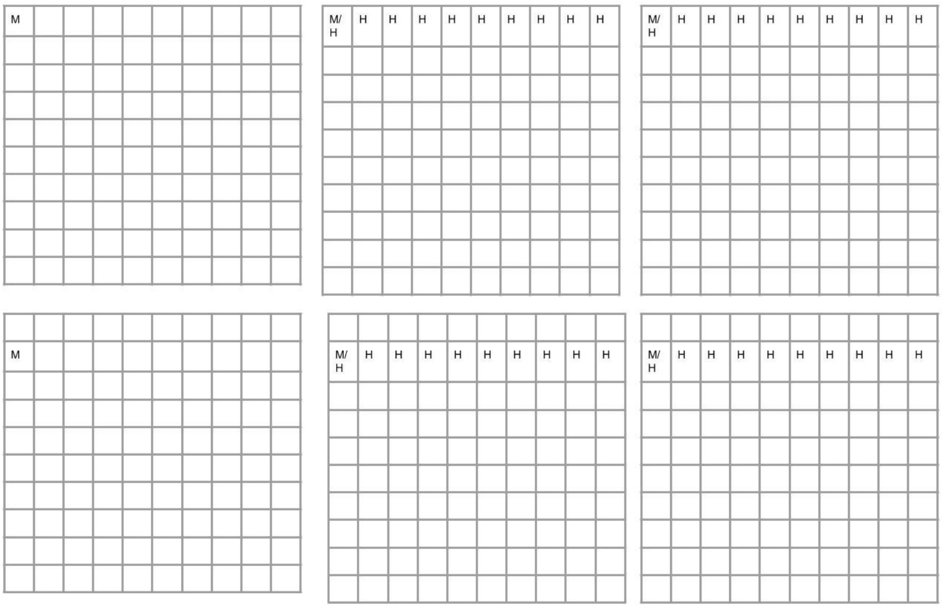
As the result:

Read cache miss rate of Matrix_A = 10/1000 = 1/100

Read cache miss rate of Matrix_B = 10/1000 = 1/100

Read cache miss rate of Matrix_C = 10/100 = 1/10

***Algorithm kij(ikj)*:**



The first time to access the head (first element of each row) of matrix_B and matrix_C is a miss, the second time and afterward would be a hit. The matrix_A only access 10x10 time and only miss at the first element.

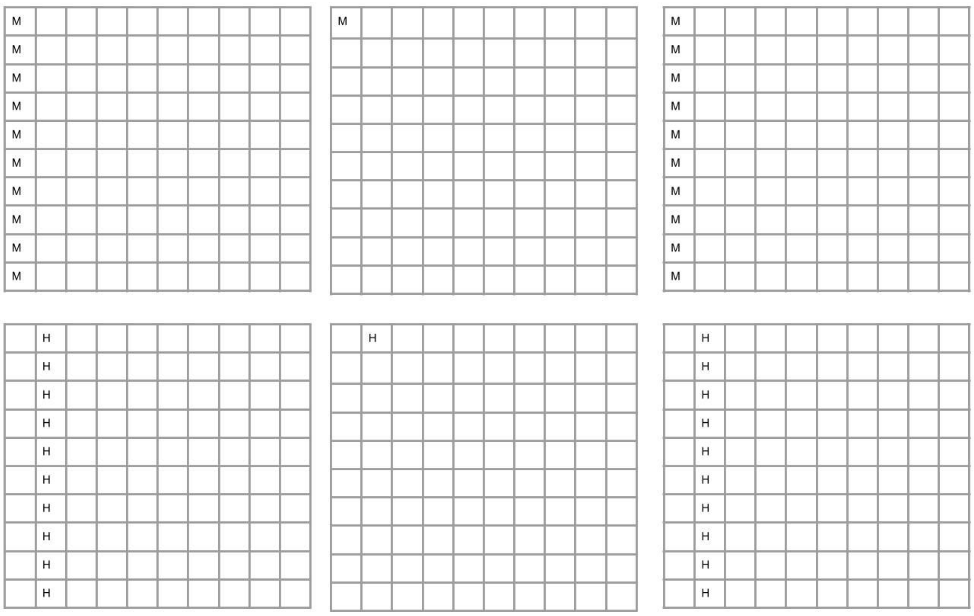
As the result:

Read cache miss rate of Matrix_A = 10/100 = 1/10

Read cache miss rate of Matrix_B = 10/1000 = 1/100

Read cache miss rate of Matrix_C = 10/1000 = 1/100

***Algorithm jki(jki)*:**



Since we start from the first column of matrix_A and matrix_C, the first iteration will be all miss. The 10 miss at the beginning will be the total miss out of 1000 access. Matrix_C miss at the first element of each row and hit at the rest access.

As the result:

Read cache miss rate of Matrix_A = 10/1000 = 1/100

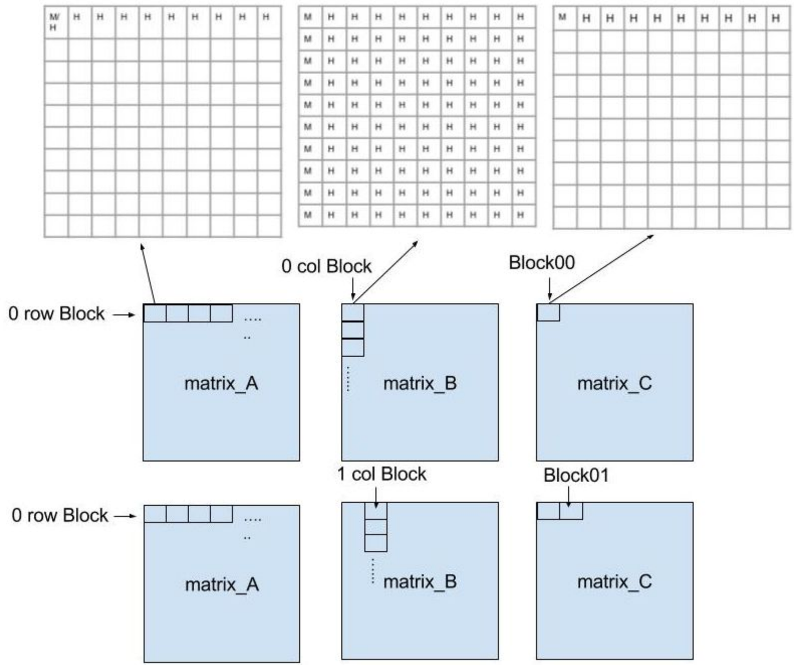
Read cache miss rate of Matrix_B = 10/100 = 1/10

Read cache miss rate of Matrix_C = 10/1000 = 1/100

Part #2

Every iteration takes (n/B) times B*B mini matrix multiplication. The miss count and miss rate of B*B mini matrix multiplication is as Part #1. Since the size of cache line (10 double nodes) couldn’t cover the next mini-block, every B*B mini multiplication can be regarded as independent task. The read cache miss rate = (cache miss rate in B*B mini multiplication)x[(n/B)^3]/[(n/B)^3] ((n/B)^2 iteration totally)

Algorithm ijk(jik):



The inner loop is the same as 10x10 ijk(jik). Matrix_A and matrix_C miss at the first element and hit at rest reading. Matrix_B load the whole block in the first iteration. However, the cache is not enough to obtain more than two sets of block multiplication. The miss rate keep the same in every block multiplication.

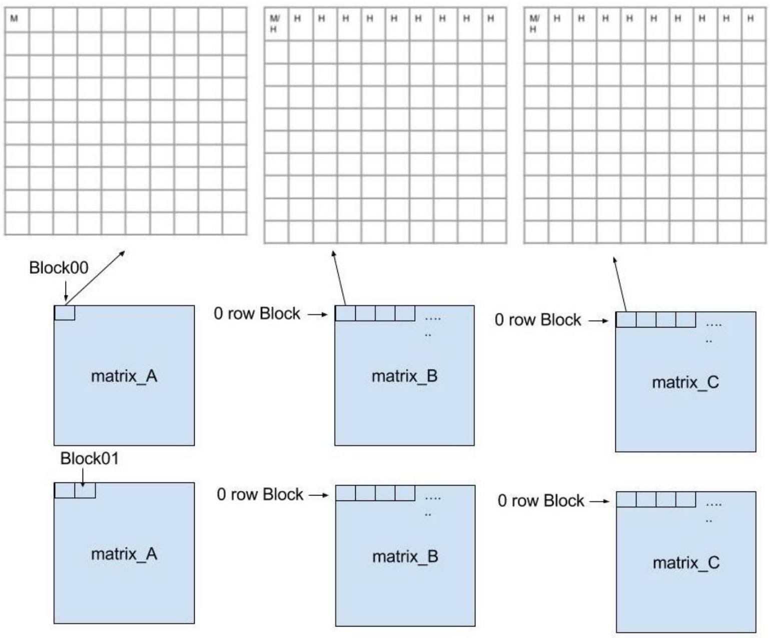
As the result:

Read cache miss rate of Matrix_A = 1/100

Read cache miss rate of Matrix_B = 1/100

Read cache miss rate of Matrix_C = 1/10

Algorithm kij(ikj):



Below each block multiplication, the first element of each row is miss while the rest reading is a hit for matrix_B and matrix_C. For matrix_A, the first reading of the first element in the row_0 is always a miss while the rest reading is hit.

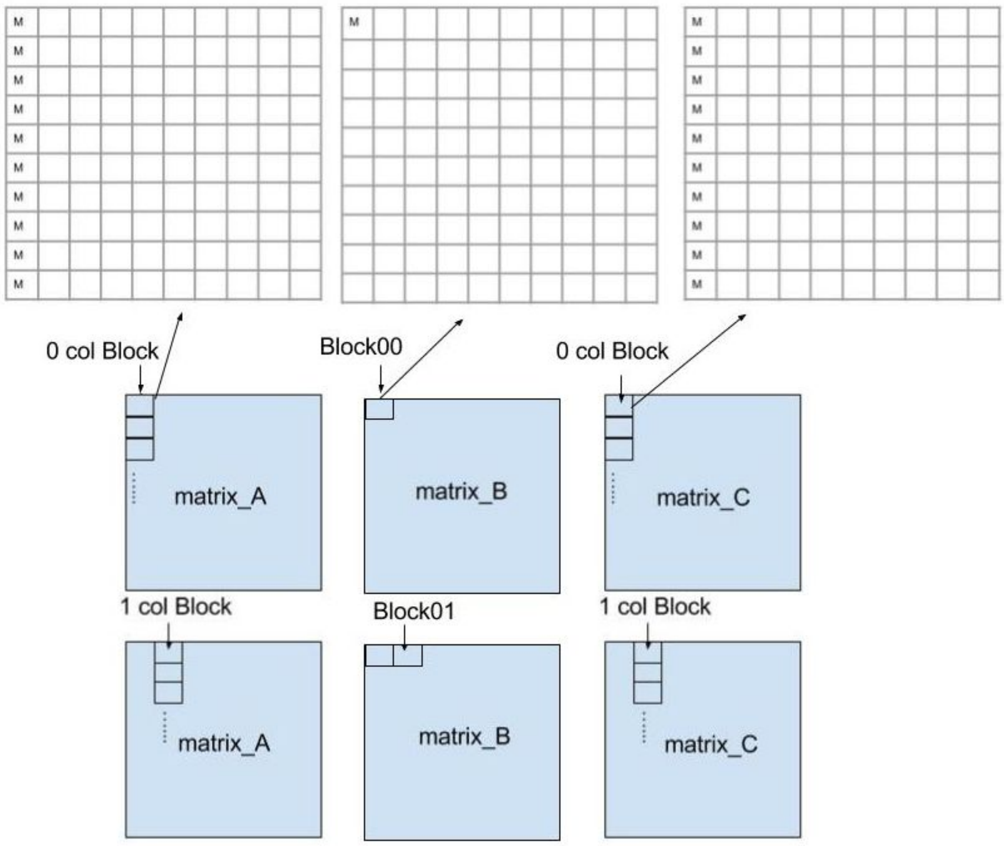
As the result:

Read cache miss rate of Matrix_A = 1/10

Read cache miss rate of Matrix_B = 1/100

Read cache miss rate of Matrix_C = 1/100

Algorithm jki(jki)**:**



Below each block multiplication, matrix_A and matrix_C miss in every first reading of first row and hit on the rest reading. Matrix B read the first element of the row with a cache miss and hit at the rest elements.

Read cache miss rate of Matrix_A = 1/100

Read cache miss rate of Matrix_B = 1/10

Read cache miss rate of Matrix_C = 1/100

Part #3

Compare with the non-blocking version and change block size. (n = 2048, B = 10 and 2^n) Here is the result of different blocking policy on 6 algorithm (ijk, kji, kij, ikj, jki, kji). Turn out we found out that the best performance comes out when block size is 64 (sometime in 32).

The best performance is approximately 0.15 Gflops.

Ijk_jik (block_1_2_A_Result.o220381)

Block Size	non-blocking	10	2	16	32	64	256	512
Time ijk/ jik	325627.687	125981.562	241584.125	123307.906	113556.75	111124.734	173322.859375	209591.015625
	311259.5312	311459.406	267758.843	126704.703	118811.367	173322.859	175537.703125	191358.203125
Gflops	0.052699	0.136368	0.071113	0.139325	0.15128*	0.150961	0.099121	0.081969

ijk/ jik	0.055195	0.055159	0.064162	0.135590	0.14459*	0.099121	0.097870	0.089779

(millisec)
(Gflops)

Kij_ikj (block_1_2_B_Result.o220382)

Block Size	non-blocking	10	2	16	32	64	256	512
Time kij/ ikj	117167.765625	121948.421875	167272.921875	124422.023438	112103.171875	114843.617188	124016.554688	150114.109375
	123509.179688	124537.156250	209187.437500	130613.851562	111607.218750	110066.734375	129878.234375	152788.5
Gflops kij/ ikj	0.146626	0.140878	0.102706	0.138077	0.15325*	0.149594	0.138529	0.114445
	0.139098	0.137950	0.082127	0.13153	0.153932	0.15608*	0.132277	0.112442

(millisec)
(Gflops)

Jki_kji (block_1_2_C_Result.o220383)

Block Size	non-blocking	10	2	16	32	64	256	512
Time jki/ kji	470238.968750	553099.000000	320325.843750	169816.437500	156432.468750	144987.046875	258064.468750	279454.312500
	559186.312500	568340.125000	351080.687500	178537.531250	167320.484375	158918.421875	240901.593750	249569.765625
Gflops jki/ kji	0.036534	0.031061	0.053632	0.101167	0.109823	0.11849*	0.066572	0.061476
	0.030723	0.030228	0.048934	0.096226	0.102676	0.10810*	0.071315	0.068838

(millisec)
(Gflops)

Part #4

We get a better performance by using both blocking cache reuse and register reuse n = 2048. Since the result from Part#3 shows that the best performance comes out when block size = 32 and 64, we choose cache block = 32 and register block = 2 to handle matrix n = 2048. Compare the different performance when using different optimization flags, from -O0 to -O3 (-O0 should be the same as original gcc)

The best performance is 1.8877 Gflops when cache blocking size = 32.

Block size = 32 (block_both_Result.o220658/ block_both_O0_Result.o220659/ block_both_O1_Result.o220660/ block_both_O2_Result.o220661/ block_both_O3_Result.o220662/)

	non-blocking	blocking	N-blocking -O0	Blocking -O0	N-blocking -O1	Blocking -O1	N-blocking -O2	Blocking -O2	N-blocking -O3	Blocking -O3
Time	300989.718750	35631.371094	315209.343750	38304.308594	255195.515625	11350.170898	322321.500000	10120.150391	374927.125000	9100.90527*
Gflops	0.05707	0.48215	0.05450	0.44851	0.06732	1.51362	0.05330	1.69759	0.04582	1.8877*

(millisec)
(Gflops)

Block size = 64

	non-blocking	blocking	N-blocking -O0	Blocking -O0	N-blocking -O1	Blocking -O1	N-blocking -O2	Blocking -O2	N-blocking -O3	Blocking -O3
Time	350159.718750	42349.851562	319723.000000	41824.089844	427432.218750	10911.872070	351372.843750	10843.834961	544019.250000	9287.662109
Gflops	0.04906	0.40566	0.05373	0.41076	0.04019	1.57442	0.04889	1.58429	0.03158	1.84975

(millisec)
(Gflops)